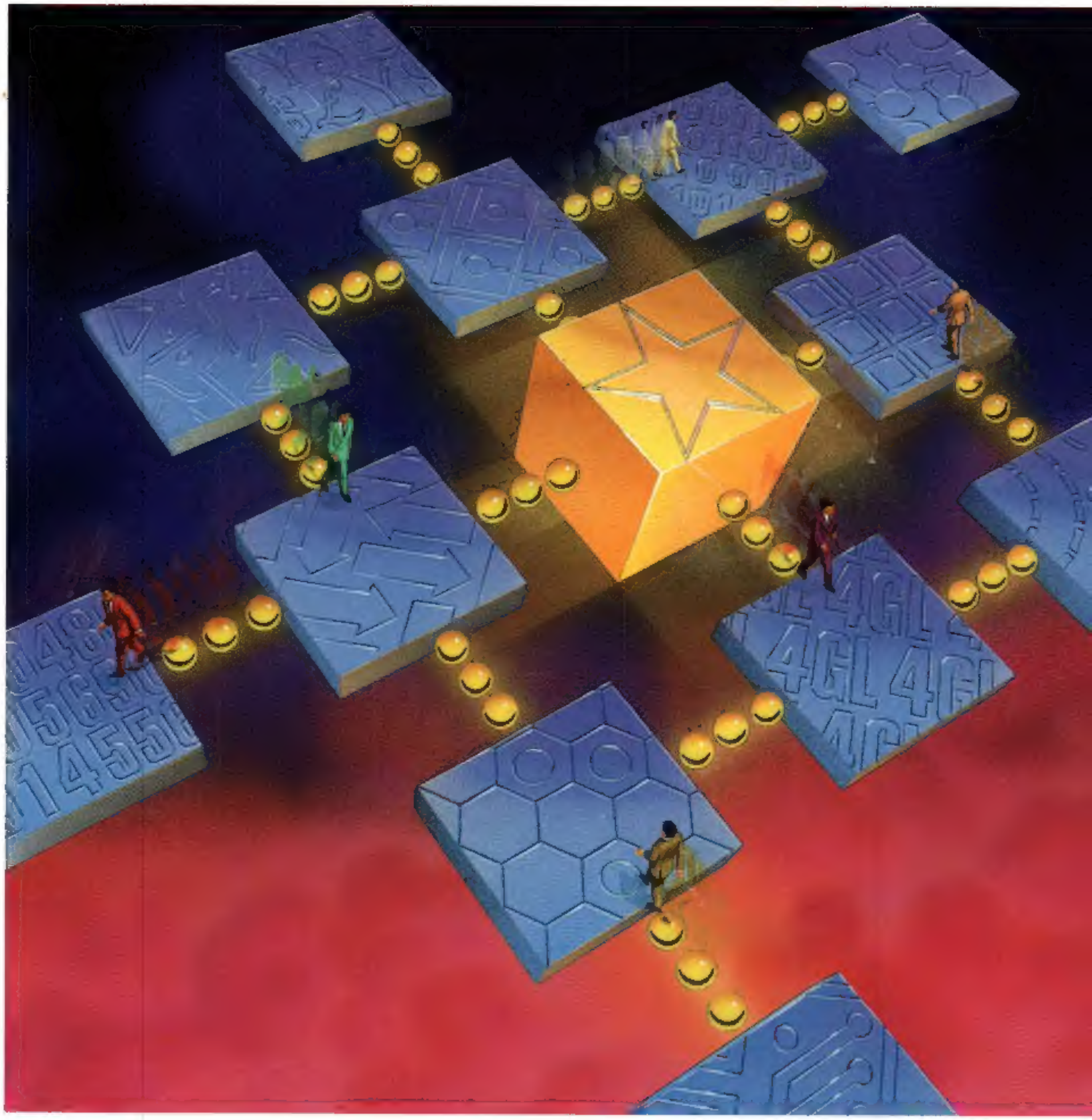
APL★PLUS® II/386



VERSION 4

User Manual

STSC

APL★PLUS® III/386

VERSION 4

User Manual

STSC

Contents

Introduction

This *User Manual* and its accompanying *Reference Manual* document the APL★PLUS II System for the 80386, an APL application development system. It is primarily directed toward APL programmers, although APL novices may find the system useful for learning APL.

If you are not familiar with APL, read an introductory text, such as *APL Is Easy!* (STSC, 1991) or *APL: An Interactive Approach* by Leonard Gilman and Allen J. Rose (Wiley, 1984). Both books are available from STSC.

Organization of the Manual

This *User Manual* is organized by topic, from basic to advanced. Early chapters describe the system and tell you how to use the session manager. Later chapters concentrate on the advanced features of APL★PLUS II/386 and tell you how to use them in your own applications.

- Chapter 1 tells you how to get started using your system.
- Chapter 2 describes the session manager and explains how you use the pull-down menus, create and manipulate windows, and how you edit objects or native files.
- Chapter 3 explains how you use the file system.
- Chapter 4 describes how you format data using the system function ⎕*FMT*.
- Chapter 5 explains how you use manage screen and keyboard data, record and play back keystrokes, and use color.
- Chapter 6 explains how you use ⎕*WIN* and related system functions to develop full-screen applications.
- Chapter 7 describes the evolution of APL★PLUS II/386 and explains how you use advanced APL language features.
- Chapter 8 describes the system's graphics capabilities.
- Chapter 9 describes the communications facilities. It explains how you use the built-in terminal mode and you use ⎕*ARBIN* to communicate with remote devices. This chapter also describes the data transfer facilities.
- Chapter 10 describes exception handling — a technique that allows your programs to react to errors.
- Chapter 11 describes the advanced features of the system, including debugging techniques, interfaces to non-APL programs, and the system customization techniques.

- Chapter 12 explains how to use the ⎕*NA* facility to use external routines from APL★PLUS II/386. It explains how to create and use associated functions, discusses the differences between real mode and protected mode, and describes the tools you can use to build externally resident modules.

Syntax Conventions

When possible, this manual shows examples exactly as you see them on your computer. It shows user entries with the standard APL six-space indent. If an example is too wide to fit on the page, it is wrapped to the next line. Unless otherwise stated, all of the examples assume an index origin of 1.

Items in *APL font* represent actual system output or information that you should enter exactly as shown. Items in *lowercase italic* font represent either information that you supply, or system output that varies. Items in `Courier font` represent information displayed by DOS or information that you enter in DOS.

1 Getting Started

The APL★PLUS II System for the 80386 is an APL application development system designed especially for the 80386 hardware environment. It merges APL microcomputer technology with features of APL minicomputer and mainframe technology to produce an APL system that is fast, powerful, and efficient.

The system has two layers, which are combined into a single program: the interpreter and the session manager:

- the interpreter is the engine that executes APL statements and programs
- the session manager is the user interface that allows you to enter and edit data and programs, and to control your hardware.

Before continuing with this chapter, install APL★PLUS II/386 on your computer. Be sure to follow the instructions in the *Installation Guide,* which contains important information on setting the necessary environment variables, adjusting the CONFIG.SYS file, and selecting and loading an appropriate APL font.

Once the system is installed, be sure you read the READ.ME file on the APL Systems disk. It contains the latest information on the system. To print the file, insert the Systems disk into the A disk drive and type:

```
type a:read.me >prn
```

Configuring the System

You can use session startup parameters to customize APL★PLUS II/386 to suit your needs. If you use the same startup parameters for each session, you can store them in a configuration file. This configuration file contains all of the parameters for an APL session, and sets the default values for an APL session. With this file, you can:

- specify the workspace to be loaded when you enter APL
- define default libraries
- select the type of keyboard you want to use
- set up other parameters and options.

A sample file, called the CONFIG.APL file, is supplied with the system. You can use any DOS text editor to customize this file to suit your needs, or you can use APL to edit the file.

To have APL automatically search for the correct file, set the following DOS environment variable in the AUTOEXEC.BAT file:

```
        set aplconfig=fileid
```

The *fileid* should be a fully qualified file identification, including the disk drive, path, and file name; for example:

```
        set aplconfig=c:\apl\config.apl
```

When you start APL, it uses the file named in this environment parameter. If you do not specify a filename in the AUTOEXEC.BAT file, APL searches for a file named CONFIG.APL in the default directory.

APL uses all of the parameters specified in the configuration file. You can specify other configuration files with the `config=` parameter; when the system reaches the line calling another file, it reads that file and sets those parameters. The system can read up to 15 separate files.

You can specify startup parameters or suppress the use of the default configuration file (if any) on the DOS command line when you start APL.

You can specify multiple values for the `key`, `lib`, `pfkey`, and `poke` parameters. For all other startup parameters, you can specify only one value. If you specify multiple values for the same parameter in different files or on the command line, the system uses the last value it reads.

If the system does not find the parameter settings, it uses the default values listed in the descriptions below.

Each line of the file contains one parameter of the form:

Parameter=value

For many parameters, you can use the mnemonic form or a one-letter abbreviation (where compatible with the APL★PLUS PC System). The parameter descriptions below use the mnemonic; uppercase letters in the mnemonic show the abbreviation. If the mnemonic does not contain an uppercase letter, you must use the whole mnemonic. The case of the mnemonic is irrelevant.

- **`aplattrs=`*n m***
  Default: 7 for the APL session
  112 for the status line

  Use this parameter to specify colors for APL session and status line. (The color for the APL session is also used in terminal mode.)

- **`blink=`*on/off***
  Default: on

  Use this parameter to turn blinking on or off. If `blink=off`, EGA and VGA adapters interpret the high-order bit of the attribute byte as high-intensity background color rather than blinking foreground color. **Note**: Use only if your system supports blink switching.

- **capslock=*on/off***
  Default: current DOS state

  Use this parameter to specify whether CapsLock is on or off.

- **cgisize=*n***
  Default: 0

  Use this parameter to specify the number of kilobytes of DOS memory (below 640K) you want the system to reserve for GSS*CGI graphics.

- **Commbuff=*n***
  Default: 4

  Use this parameter to specify the serial port communications buffer size in kilobytes. The range is 0 to 63.

- **config=*fileid***
  Default: none

  Use this parameter to specify another configuration file by name. The system reads this file upon recognizing this entry, then continues with the current file.

- **dragdown=*on/off***
  Default: off

  Use this parameter to specify whether to activate "dragdown" behavior. Dragdown lets you edit a line, transfer it to the bottom of the screen, and execute it. The original line is restored. (See the "Using the Keyboard and Mouse" section later in this chapter.)

- **duplex=*full/half***
  Default: half

  Use this parameter to specify full or half duplex for terminal mode.

- **`editattrs=`*n m***
  Default: 7 for an edit session
  112 for status line

  Use this parameter to specify colors for edit session and edit status line.

- **`editnums=`*on/off***
  Default: off, do not display line numbers

  Use this parameter to specify whether line numbers are displayed when starting a new CV, CM, or NF editing sessions. (`editnums=` does not affect numeric or function editing sessions, where line numbers are always displayed.)

- **`edittype=`*cv/cm/fn***
  Default: CV

  Use this parameter to specify the type of editing session created when editing an undefined name.

- **`Env=`*n***
  Default: 0 (IBM)

  Use this parameter to specify the environment (host hardware and operating system). See the Advanced Techniques chapter in this manual for details. The default is 0 (IBM). Use these values.
  - ❑ `0`
    Fully IBM compatible.
  - ❑ `2`
    Fixed DOS BIOS keyboard for input.
  - ❑ `8`
    Beep.
  - ❑ `16`
    COM1 RS-232 serial port.
  - ❑ `32`
    LPT1 printer interface.

- ❑ `256`
  Buffer COM2 and use COM2 for terminal mode instead of COM1.

- ❑ `1024`
  Resident driver for soft character set support.

Values `16` and `256` are often used when a mouse is on COM1 or a bus mouse is installed.

- ■ **`evlevel=`*n***
  Default: 1

  Use this parameter to specify the evolution level that controls the behavior of evoloving language features. See the Using Advanced Language Features chapter for more details.

- ■ **`Graphloc=`*n***
  Default: none

  Use this parameter to force the `graphmem=` parameter to use printer graphics memory at a specific kilobyte location below 1024K. You must have arranged for memory to be available at this location. `Graphloc` generally provides no added value to systems using standard hardware and current operating systems.

- ■ **`gRaphmem=`*n***
  Default: 0

  Use this parameter to specify the size of memory in kilobytes for use by `⎕GINIT 'PRINTER'`. The system finds an available region above 1024K and uses it for this memory, unless you specify the memory location with the `gRaphmem=` parameter. See the Using Graphics Capabilities chapter for more information.

- ■ **`help=`*fileid***
  Default: SYSHELP

  Use this parameter to specify the name of an APL component Help file. If the Help file you specify is not in the current DOS directory, be sure you include the DOS pathname with the fileid. **Note:** Be sure to set this parameter if you plan to use a

Help file located in a directory other than the current directory.

- **hercsoftchr=*n***
  Default: 0

  Use this parameter to specify if a Hercules graphics adapter is in use. This is only necessary when you use the e=1024 graphic character set. On (1) specifies Hercules.

- **Holdbuf=*n***
  Default: 65534, requires 2 bytes per character

  Use this parameter to specify the number of characters that can be stored in the ⎕*INBUF* buffer.

- **ignorecase=*on/off***
  Default: off, do not equate upper and lowercase

  Use this parameter to specify whether the system should ignore case when it performs a search using Ctrl-L or Ctrl-R.

- **initWs=*wsid***
  Default: clear workspace

  Use this parameter to specify the initial workspace you want the system to load.

- **insert=*on/off***
  Default: on

  Use this parameter to specify insert or replace mode.

- **key *n=value***
  Default: none

  Use this parameter to redefine the keyboard table by assigning a given event number to a given keybutton number. See the Advanced Techniques chapter for details.

- **`Keyboard=`*text/APL***
  Default: APL

  Use this parameter to specify the keyboard layout. Valid entries are Text or APL.

- **`keylog=`*'fileid R'***
  `keylog=`*fileid*
  Default: none

  Use this parameter to record all keystrokes into the file specified. The *R* is optional and causes the system to replace the contents of an existing file with the new keystrokes.

- **`keysrc=`*'fileid n'***
  `keysrc=`*fileid*
  Default: none

  Use this parameter to specify the input file from which the system obtains keystrokes. *n* is an optional even non-negative integer that specifies the offset in the file where you want ⎕*KEYSRC* to begin reading keystrokes.

- **`lib` *n=path***
  Default: none

  Use this parameter to define a library. The *n* is the library number; *path* is the directory you want to define as a library.

- **`Linefeed=`*on/off***
  Default: on

  Use this parameter to indicate whether linefeeds are sent to the printer after carriage returns. On sends linefeed; off suppresses linefeed.

- `linelog=`*'fileid R'*
  `linelog=`*fileid*
  Default: none

  Use this parameter to record session input and output into the file specified. The R is optional and causes the system to replace the contents of an existing file.

- `mouse=`*on/off*
  Default: on, use mouse if present

  Use this parameter to specify whether the system should use or ignore the mouse. When `mouse=off`, force system to ignore a mouse that is present.

- `Network=`*on/off*
  Default: on, if SHARE was run
  off, if SHARE was not run

  Use this parameter to indicate whether the system should use an available network.

- `NOCONfigapl`
  Default: none

  Use this parameter to suppress reading of the default configuration file; use on DOS command line only.

- `numlock=`*on/off*
  Default: DOS state on entry

  Use this parameter to indicate whether the system should use the numeric pad for numbers or cursor movement.

- `pfkey` *n=values*
  Default: none

  Use this parameter to specify the values for PFkey *n*. (See the "Working in an APL Session" section later in this chapter.)

- **pfmem=*n***
  Default: 2048

  Use this parameter to specify the number of bytes you want the system to reserve for function key storage; the minimum is 1000.

- **pfnum=*n***
  Default: 127

  Use this parameter to specify the maximum number of concurrently defined function keys.

- **poke *loc=value***
  Default: none

  Use this parameter to set a system parameter, controlled by ⎕*POKE*, to a specific value. See the Peeks and Poke appendix in the *Reference Manual* for settings and restrictions.

- **preserved=*n***
  Default: none

  Use this parameter to preserve *n* lines above the screen in the main session to prevent deletion during editing sessions.

- **Printerport=*n***
  Default: none

  Use this parameter to specify system printer port. The default is LPT1 (3). (See "Printing Data.") Valid values are:

1	COM1
2	COM2
3	LPT1
4	LPT2
5	LPT3
10	Int90h driver

- `Qinitws=`***wsid***
  Default: none

  Use this parameter to specify the workspace you want the system to quiet load.

- `Screenmem=`***n***
  Default: 80

  Use this parameter to specify the amount of memory, in kilobytes, you want the system to allocate for sessions in the session manager.

- `sendctrl=`***on/off***
  Default: on (send)

  Use this parameter to indicate whether the system should send Ctrl- keystroke combinations in terminal mode or perform the appropriate local action. (See the Communications chapter for more information.)

- `searchtype=`***strings/tokens***
  Default: strings

  Use this parameter to specify whether the system should search for strings or tokens.

- `statline=`***on/off***
  Default: on (display)

  Use this parameter to specify whether the system should display or suppress the status line.

- `tagattrs=`***n m***
  Default: 15 for a partially tagged region
  120 for a fully tagged region

  Use this parameter to specify colors you want the system to use for a partially tagged region and for a fully tagged region.

- **tempdir=*directory***
  Default: current directory on current disk drive

  Use this parameter to specify the directory for the temporary file that *⎕COPY* and *)COPY* create.

- **termmode=*apl/ascii***
  Default: APL

  Use this parameter to specify whether the system should use APL overlay or ASCII transmission for terminal mode.

- **termstyle=*vt/dm***
  Default: dm

  Use this parameter to specify whether the system should use VT100 (vt100 or vt) or Datamedia 1520 (dm1520 or dm) terminal emulation for terminal mode

- **Translation=*n***
  Default: 2 (ASCII)

  Use this parameter to select the system printer translation. See the Communications chapter in this manual and the "*⎕ARBIN*" section in the *Reference Manual* for other values.

- **ucmdfile=*name***
  Default: the name of the current directory at startup, followed by UCMDS

  Use this parameter to specify the name of the default APL file that contains user commands. For example, C:\APL\UCMDS.

- **user=*n***
  Default: 1

  Use this parameter to indicate the ID number of the user.

- **wrapattr=*n***
  Default: 1

  Use this parameter to specify the color the system should use for the wrapped-line marker.

- **wssize=*n***
  Default: the maximum number of kilobytes permitted by your hardware

  Use this parameter to specify the size of the active workspace in kilobytes.

- **wstate *position=value***
  Default: none

  Use this parameter to set an element of ⎕*WSTATE* to the value specified. Positions are in origin 1.

- **x=*n***
  Default: none

  Use this parameter to specify a user-defined parameter 1 (accessible with ⎕*PEEK* 276). You must specify an integer between 0 and 255.

- **y=*n***
  Default: none

  Use this parameter to specify a user-defined parameter 2 (accessible with ⎕*PEEK* 277). You must specify an integer between 0 and 255.

- **z=*n***
  Default: none

  Use this parameter to specify a user-defined parameter 3 (accessible with ⎕*PEEK* 278). You must specify an integer between 0 and 255.

Working in an APL Session

An APL session is more than an active workspace. This section tells you how to start and end an APL session, gives you an overview of the session manager, describes the status line, and tells you how to get on-line Help during an APL session.

Starting and Ending an APL Session

To start an APL session, at the DOS prompt, type the APL command:

```
apl386
```

If your computer has an 80486DX processor, or an 80487SX, 80387, or 80287 co-processor, type the APL command:

```
apl387
```

Although you can use the APL command `apl386`, the command `apl387` gives faster performance on computers that have a numeric data processor.

Be sure that you are in the APL directory or that you have a path to it. The system first searches for a configuration file and sets up the session according to the parameters you specified. The default configuration file is the one specified in the AUTOEXEC.BAT file by the `aplconfig=` environment variable. If you do not specify a configuration file, the system looks for the CONFIG.APL file in the current directory.

You can also specify parameters and configuration files on the same line as the APL command. The system reads and executes settings or configuration files from left to right. The last setting or file read is the one that remains in effect; however, you can specify several values for the `key`, `lib`, `pfkey`, and `poke` parameters. The system uses all of the values you enter for these parameters. For example, if you do not have an initial workspace specified in a configuration file, you can specify one on the command line:

```
        apl387 initws=c:\apl\myws
```

If you specified an initial workspace in a configuration file, the one specified at the end of the command line overrides that parameter.

To suppress the use of the default configuration file, use the `noconfigapl` (or `nocon`) parameter anywhere on the command line. If the system finds this parameter, it reads only configuration files explicitly specified by the `config=` parameter.

Once the system reads any specified or default configuration files, the APL session begins.

To end an APL session, type the APL command:

```
        )OFF
```

APL ends and the DOS prompt appears. Any work that you have not saved to disk is discarded.

Overview of the Session Manager

The session manager is the APL★PLUS II/386 user interface. It allows you to have a workspace and several full-screen editing sessions active simultaneously. You begin APL in an APL session, which is the active workspace. You can then create new editing sessions, where you can define functions or edit arrays and native files.

Think of the sessions as being arranged in a ring. You can move around the ring in either direction or move directly to the APL session from any editing session. You can also go into terminal mode, where you use your personal computer as a terminal to another computer. (See the Communications chapter for more information.)

You can use either pull-down menus or keystroke combinations to set or change options, perform common editing tasks, and move around or among sessions.

You can also use the session manager to move pieces of text among sessions or to move data and functions between workspaces. For example, you can edit different functions in several concurrent editing sessions. Rather than typing the same code in several functions, you can select (or tag) that piece of code and copy it into all of the functions that use it. You can also tag code in the APL session and move it into a function.

The Editing with the Session Manager chapter describes the capabilities of the session manager in detail. The next section describes basic techniques you can use to enter input and to edit your input.

The Status Line

The status line at the bottom of your computer's screen displays the current value of certain keyboard- and workspace-related parameters:

- the name of the current session or object
- whether terminal mode is active
- the keyboard state (APL or Text)
- whether CapsLock On has been selected
- whether the cursor or numeric pad has been selected
- whether Insert or Replace mode has been selected
- whether the Index Origin (⎕*IO*) = 0
- whether the printer is on
- whether Log is in effect
- a nonempty state indicator
- the cursor position (row and column) within the session
- the type of object being edited
- whether lines are tagged in this or other sessions
- the type of translation used in terminal mode
- whether Ctrl- keys are sent in terminal mode or act locally.

You can turn the status line off by pressing ScrollLock.

Because each type of session has a unique status line, you can easily determine which session you are in.

Getting Help

APL★PLUS II/386 has a built-in Help facility that gives you on-line information about various system features. This section explains how to start and use the Help facility.

Starting Help

The Help facility comes with a default help file, SYSHELP.SF. To use the Help facility from any directory, set the `help=` startup parameter; otherwise the system looks for SYSHELP in the current directory. To start Help from an APL session, type:

```
      )HELP
```

Or, to start Help from within a user-defined function, type:

```
      ⎕HELP ''
```

The system accesses the Help facility and displays the Main Help menu.

You can also specify the topic you want help on when you start Help. To display information on a specific topic, type

```
      )HELP topic
```

or

```
      ⎕HELP 'topic'
```

You can press Alt-F6 to use the Help facility from any session. If Help finds the topic you specify, it displays the information on the screen. Otherwise, Help displays a message that informs you that it could not find the topic you asked for and ends the Help session. You can either specify another topic or start Help without specifying a topic and manually search through Help for the information you want.

Displaying Information in Help

Once you are in the Help facility, the cursor is at the bottom row of the screen and the system waits for your input. You can select and display information using either the keyboard or a mouse. The following list describes how to move around in Help.

- To select an item from a menu or submenu, use one of these methods.
  - Use the arrow keys on the keyboard to position the cursor on the item you want and press Enter.
  - Position the mouse pointer on the item and click the left mouse button.
  - Type the number of the menu item.
- To display information about a topic, type the name of a topic and press Enter. You can enter up to 24 characters. You can get help on the following topics from any screen in Help.
  - APL primitive functions. Enter the function's symbol; for example, typing rho (⍴) displays information on the Shape/Reshape function.
  - System functions and commands. Enter the system function or command; for example, ⎕*ARBIN*.
  - Peeks and Pokes. Enter P followed by the peek or poke you want; for example, P110.
  - Error messages. Enter the message as it appears on the screen.

- To display the previous screen, use one of these methods.
  - Press Alt-F10.
  - Click the right mouse button.
- To exit Help, use one of these methods.
  - Press the Esc key.
  - Click both mouse buttons at once. (On three-button mice, click the two outside buttons.)

Using the Keyboard and Mouse

This section explains how you use the keyboard and mouse. It describes the two keyboard states, and how you enter and edit information in the system.

Keyboard States

APL★PLUS II/386 allows you to use the keyboard in two basic states: as the APL keyboard or as a text keyboard.

- **APL Keyboard State**
  The APL Keyboard State extends the traditional APL keyboard. The unshifted letter keys display the uppercase APL letters; shifted letter keys display APL symbols; and Alt-letter combinations display lowercase APL letters and composite APL symbols. Alt-Shift combinations provide the punctuation keys and ASCII characters on the top row.

- **Text Keyboard State**
  The Text Keyboard State provides a combination of the ASCII character set and APL characters. This state retains ASCII character positions. The unshifted letter keys display lowercase; shifted letter keys display uppercase; Alt-letter combinations display simple APL symbols; Alt-Shift-letter combinations display composite APL symbols.

The default state is the APL keyboard. You specify the text keyboard with the `Keyboard=` option in the CONFIG.APL file. To switch between keyboard states, select the appropriate option from the Keyboard Menu or press Alt-CapsLock.

The numeric pad on the keyboard can also be in one of two states: cursor or numeric.

- **Cursor State**
  When the pad is in cursor state (the default), you use the unshifted keys to move the cursor around the screen; you use shifted keys to display digits.

- **Numeric State**
  When the pad is in numeric state, you use unshifted keys to display digits; you use shifted keys to move the cursor.

To change the pad state, select the appropriate option from the Keyboard Menu or press NumLock.

In addition to the two keyboard states, you can use three possible keyboard drivers, which capture and interpret various keystroke combinations: the APL keyboard driver, the switchable enhanced DOS keyboard driver, or the fixed enhanced DOS keyboard driver.

- **APL Keyboard Driver**
  The APL keyboard driver is the default. It provides a larger and more flexible set of Shift key combinations than are possible with DOS keyboard handling facilities.

- **Switchable Enhanced DOS Keyboard Driver**
  For information on the switchable enhanced DOS keyboard driver, see the "How to Use the Function Keys" section in this chapter, and the "DOS Keyboard Handling" section in the Advanced Techniques chapter.

- **Fixed Enhanced DOS Keyboard Driver**
  For information on the fixed enhanced DOS keyboard driver, see the "DOS Keyboard Handling" section in the Advanced Techniques chapter.

You may need to use one of the DOS keyboard drivers and DOS keystrokes if:

- you are running memory-resident software or other software that requires "hot keys" to activate it

- you are using a program that obtains keyboard input through a modem using BIOS or DOS keyboard calls.

Entering and Editing Information in an APL Session

Your computer's screen displays information that you type on the keyboard; the system holds the information until you press Enter. Once you press Enter, the system processes the information. Until you press Enter, you can edit the information that you typed.

The following section explains how you enter and edit information, and assumes that you are using the default keyboard mappings. Keys are identified by the marks on the physical keytop to avoid confusion between keyboard states. For example, Ctrl-+ refers to the key with the = and + symbols on it, not the key that displays the APL + symbol in the APL keyboard state.

- **Scrolling**
  Scrolling is the process of moving the window up and down, or left and right through the stored screen images. To understand the scrolling capabilities of the session manager, think of your computer screen as a window that displays only part of the input and output generated during the course of an APL session. The session manager maintains a record of recent output. It also stores additional lines of information that appear after the screen is full.

  Vertical scrolling occurs automatically when all of the lines on the screen are used and a new line of output is produced. All of the lines on the screen move up one position, and the topmost line disappears. If you move the cursor to the top line of the screen and press the up arrow key (↑), you "push" the window up one line to reveal the line that disappeared. Similarly, pressing the down arrow key (↓) when the cursor is at the bottom of the screen pushes the window down one line.

  If you are at the top of the scrolling memory and try to move up, the session manager beeps. The same effect is produced when you attempt to move past the last line with a cursor key. The memory used to record the session log is allocated from the same pool of storage that is used to hold objects in the editing sessions. When all of this memory is in use, the system discards the lines at the top of the APL session log as it adds new lines. By default, the system allocates 80 kilobytes for session and editor memory.

Horizontal scrolling lets you display lines of information that are longer than your computer screen is wide. If you move the cursor to the right edge of the screen and press the right arrow (→) key when horizontal scrolling is in effect, the entire screen display shifts right one character. To activate horizontal scrolling, press Crtl-Shift-H. You can also activate horizontal scrolling by setting `wstate=31` in the configuration file or executing the statement `⎕WSTATE[1]←1` during an APL session.

- **Moving the Cursor and Window**
  Use the cursor keys (up, down, left, and right arrow keys — ↑ ↓ ← →) to move up and down one line at a time or left and right one position at a time. The Tab and Backtab keys move right and left one tab stop at a time. The Home key moves the cursor to the first nonblank character on a line; End moves the cursor one position past the last nonblank character on a line. The PgUp and PgDn keys move the window up or down one row.

  You can press the control key (Ctrl) with any of the cursor movement keys described above to magnify the effect:

  - Ctrl with the up arrow (↑) and down arrow (↓) keys moves the cursor up or down four rows.

  - Ctrl with the left arrow (←) and right arrow (→) keys moves the cursor left or right eight positions.

  - Ctrl-Home moves the cursor to the beginning of the screen; Ctrl-End moves the cursor to the beginning of the last line on the screen.

❑ Ctrl-PgUp moves the window up one line less the number of lines on the screen; Ctrl-PgDn moves the window down that amount. For example, Ctrl-PgUp moves the window up 24 lines on a 25-line screen.

❑ Ctrl-Tab moves the window right one screen width.

❑ Ctrl-Shift-Tab moves the window left one screen width.

You can press the Alt key with the following keys to take the movements to their limits:

❑ Alt with the up arrow (↑) and down arrow (↓) keys moves the cursor to the first or last line in a session and remains in the current column position.

❑ Alt with the left arrow (←) and right arrow (→) keys moves the cursor to the beginning or end of the current line in the session.

❑ Alt-Home moves the cursor to the beginning of the first line in the session; Alt-End moves the cursor to the beginning of the last line in the session.

❑ Alt-PgDn moves the cursor to the top line of a clear screen at the bottom of the session.

❑ Alt-Tab moves the screen right one character.

❑ Alt-Shift-Tab moves the screen left one character.

To move the cursor a specific number of rows, follow these steps:

1. Press Shift-Esc.

2. Press the number corresponding to the number of rows that you want to move.

3. Press the up arrow (↑), down arrow (↓), PgUp, or PgDn keys. The cursor moves, or the screen scrolls, the number of times you specified.

For example, to scroll the screen up 75 lines, you press Shift-Esc, type 75, then press PgUp. To move nine lines down, you press Shift-Esc, 9, then the down arrow (↓).

You can also move the cursor and window with a mouse, if you have one attached and activated. To move the cursor with the mouse, hold the left button down. The cursor moves along with the mouse pointer (as far as the edge of ⎕WINDOW) until you release the button. To move the cursor to another point in a session, move the mouse pointer to the target position and click the left button. The cursor moves to the new position.

To scroll through a session with the mouse, place the mouse pointer on the line number in the status line. Press and hold the left button to scroll up; press and hold the right button to scroll down.

- **Inserting and Replacing**

When you type information into the computer, the system ordinarily inserts the characters at the cursor position and pushes the rest of the characters to the right. This default behavior is called insert mode. Replace mode causes characters you type to overwrite any characters to the right. To switch between insert and replace mode, press the Ins key.

As you type, lines wrap to the next line; the first character of the continued line is marked with a special attribute to indicate the wrap. A wrapped line can be as long as 16383 characters and still be treated as a single line of input. You can wrap and unwrap lines explicitly.

- Place the cursor on the second line and press Alt-F1 to turn two separate lines into one wrapped line.
- Place the cursor on the wrapped part of a line and press Alt-F2 to split one wrapped line into two distinct lines.

You can also insert and delete lines. To insert a line, press Alt-F3 or Ctrl-Ins. The system inserts the line above the current line. To delete a line, press Alt-F4 or Ctrl-Del. The system deletes the current line and closes up the rest of the lines.

■ **Forming Composite Characters**

You can also form composite APL characters or overstrikes. Most of these are available directly from the keyboard, but you can also form them manually.

To enter a composite character, type the first character of the composite, then press Alt-Ins. The cursor changes shape and moves back over the first character. Now type the next character. If you try to form an invalid composite, the system beeps.

If you decide not to make the composite after you press Alt-Ins, press Space or Alt-Ins again.

One special triple composite is used to exit from character input mode (see the Managing Screen and Keyboard Data chapter). This composite comprises the characters *O U T* overstruck together. The *O* and *U* form a "smiling face" character, which is overstruck with the *T*.

■ **Deleting Information**

You can delete characters from the screen in several ways.

❑ Press Backspace to delete characters to the left of the cursor. In insert mode, the space is closed up with the characters to the right. In replace mode, the character is replaced by a space.

❑ Press Del to delete the character at the cursor and close up the space.

❑ Press Alt-Shift and the 5 on the numeric pad to erase a single character without closing up the rest of the text. The character at the cursor disappears and the cursor remains in place.

Erasing chunks of text is simple — you need not erase lines or large portions of sessions character by character.

❑ Press Alt-Shift-left arrow to erase a line from the cursor to the beginning of the line.

- ❑ Press Alt-Shift-right arrow to erase a line from the cursor to the end of the line.
- ❑ Press Alt-Shift-PgDn to erase all of the text in a session from the cursor to the end of the session.
- ❑ Press Alt-Shift-down arrow to erase all of the lines in a session from the line containing the cursor to the end of the session.
- ❑ Press Alt-Shift-PgUp to erase the entire session.
- ❑ Press Alt-Shift-End to erase all of the lines in an edit window from the cursor to the end of the window.
- ❑ Press Alt-Shift-Home to erase the entire edit window.
- ❑ Tag a block and press Ctrl-D to delete it. (See "Editing Techniques" in the Editing with the Session Manager chapter for details on tagging blocks.)
- ❑ Tag a block and click the left button of a mouse. A pop-up menu appears; select Delete to delete the block. (See "Editing Techniques" in the Editing with the Session Manager chapter for details on using the pop-up mouse menu.)
- ❑ Press Ctrl-Del or Alt-F4 to delete the current line.

■ **Undoing Deletions**
If you erase a line or block accidentally, you can undo the erase.

- ❑ Press Ctrl-U to undo a sequence of line erasures done with Ctrl-Del or Alt-F4. You can restore up to 20 lines. The system places restored lines on the line above the cursor.
- ❑ Press Ctrl-Shift-U to undo the last block erased with the pop-up mouse menu or the Ctrl-D, Alt-Shift Home, Alt-Shift-PgUp, Alt-Shift-End, Alt-Shift-down arrow, or Alt-Shift-PgDn keystrokes. You can restore only the last erased block. When the memory in the session manager

is full, it discards any lines stored in this manner before it discards stale lines from the top of the APL session.

- ❑ Press Ctrl-Backspace to restore the current line to its former state; that is, before you changed it.

■ **Recalling Previous Lines for Editing**

In an APL session, you can copy the current line to the bottom of the session by pressing Ctrl-Enter.

Alt-F10 allows you to recall the last line of input without retyping it. Alt-F10 also maintains a circular list of previous lines, so if you press Alt-F10 repeatedly, the system recalls earlier input lines. If you recall too many lines with Alt-F10, you can press Alt-F9 to back up through the recalled lines.

A "dragdown" option provides a convenient style of behavior for the APL session. If you move the cursor up to a previous line of input and press Enter, the system copies and re-executes the input line at the bottom of the session. If you modify the previous line, the system restores it to its original form when you press Enter and the modified form is typed in at the bottom of the session. This technique allows you to re-use previous lines of input and keeps a complete log of the session.

To toggle dragdown during a session, press Ctrl-Shift-K or use the Keyboard Menu (see the Using the Session Manager chapter). You specify dragdown with the `dragdown=on` startup parameter.

Using the PFkeys

The function keys, or PFkeys, are storage places for sequences of characters; for example, frequently used commands or functions. Pressing a function key is the same as typing the characters stored on it. If nothing is stored on a key, nothing happens when you press it. If the stored sequence ends with Enter, the system processes it when it is displayed. You can define the contents of a PFkey in three ways.

- **pfkey Startup Parameter**
  You can use the `pfkey` *n*= startup parameter to set up PFkeys in a configuration file or on the command line. The *n* is the number of the PFkey you want to set. The argument is the definition, which is a text string or list of event numbers. If the argument is a text string that contains embedded blanks or begins with a digit, enclose the argument in single quotes.

- **⎕*PFKEY* System Function**
  You can use the ⎕*PFKEY* system function to set up PFkeys specific to your application. (See the Managing Screen and Keyboard Data chapter for details.)

- **Shift-Esc Keystroke**
  You can use the Shift-Esc *pfkey* keystroke sequence to define a PFkey at any time during your session or while in the full-screen editor.

The last method allows you to "see" keystrokes that do not show on your computer's screen, such as cursor movements or Enter.

Press Shift-Esc to prepare for the definition. Then press the PFkey you want to define. The status line prompts you for the definition. As you type the definition, it appears on the status line. If you make a mistake when you type the definition, press Alt-Shift-Backspace. After you finish defining the key, press the function key again. The system stores the definition on that key for the remainder of the session or until you change it.

You can define function keys using a PFkey alone, using a PFkey with Ctrl, using a PFkey with Shift or using other keystroke combinations. APL reserves the PFkeys with Alt; they are shown in Table 1-1.

Table 1-1. Default PFkey Definitions

Key	Definition
Alt-F1	Wraps the line containing the cursor to the end of the previous line. The first character of the wrapped line is marked with a special attribute.
Alt-F2	Breaks a wrapped line containing the cursor into two separate lines.
Alt-F3	Inserts a blank line at the cursor, and pushes subsequent lines down one line. Same as Ctrl-Ins.
Alt-F4	Deletes the line with the cursor, and pulls subsequent lines up one line. Same as Ctrl-Del.
Alt-F5	Toggles the DOS keyboard passthrough. When the DOS keyboard passthrough is on, all of the keystrokes are interpreted by DOS rather than by APL. The status line displays (DOS Kb).
Alt-F6	Accesses the Help Facility.
Alt-F7	Toggles use of the graphics character set, if you started APL with `env=1024`. (See the *Installation Instructions* or Chapter 11 for details on this parameter.) If you have both color and monochrome monitors, and you used `1 ⎕POKE 199`, Alt-F7 changes between color and monochrome monitors.
Alt-F8	Switches between the local session and terminal mode.
Alt-F9	Back up through recalled lines. You can back up through up to 20 lines or 1000 characters.
Alt-F10	Recalls the last lines entered in the APL session, up to 20 lines or a maximum of 1000 characters. Also, backs up to the previous screen in the Help facility.

Customizing the Keyboard

APL★PLUS II/386 gives you the ability to redefine the keyboard translate table to suit your needs. In fact, you can completely rearrange the actions of the keys.

One way to redefine the keyboard is to use the `key=` startup parameter in the configuration file. Its form is

```
        key n=value
```

where *n* is a combination of shift states and keybutton number, and *value* is a keyboard event.

For details on how to redefine the keyboard, see the Advanced Techniques chapter.

Organizing Files and Workspaces into Libraries

Traditional APL systems use **libraries** to organize files and workspaces. A file or workspace identification is made up of a library number and name. In this APL★PLUS System, you can organize your APL component files and workspaces into libraries if you associate library numbers with DOS directories. This process is called **defining a library.** Or you can simply specify a DOS path when you need to specify a file or workspace. If you do not specify a DOS path, the system assumes the current directory on the current default disk drive.

By default, no libraries are defined when you start APL. You can place library definitions in the CONFIG.APL file or you can define them "on the fly" during the APL session.

For example, to define library 11 as the subdirectory *C:\APL\FILES* in the APL configuration file, enter the line:

```
      lib 11=c:\apl\files
```

The next time you start APL, the library is defined automatically.

You use the system function ⎕*LIBD* to define a library during an APL session:

```
      ⎕LIBD '11 C:\APL\FILES'
      ⎕LIBD '7 C:\APL\WSS'
```

Once you define the libraries, you display their definitions with the system command)*LIBS* or the system function ⎕*LIBS*.

```
      )LIBS
11 C:\APL\FILES 7 C:\APL\WSS
      ⎕LIBS
         11 C:\APL\FILES
          7 C:\APL\WSS
```

You can also reference the library number when you use file or workspace functions that require file identifications, such as ⎕*FTIE*, ⎕*FRENAME*, or ⎕*WSID*.

```
      '11 MYFILE' ⎕FTIE 2
      '11 YOURFILE' ⎕FRENAME 2
      )LOAD 7 TOOLS
7 TOOLS SAVED...
```

When you use the system functions ⎕*FNAMES* and ⎕*WSID*, the system first looks for a library definition in the current value of ⎕*LIBS*. If a definition exists, the system substitutes the library number for the path when it displays the result. If the system does not find a library definition, it displays and left justifies the full path name.

Even if you have libraries defined, you can still refer to a file or workspace with its full path name; for example:

```
      )LOAD \APL386\UTILITY
      )COPY A:TFNS FNREPL
```

If you reference an undefined library, however, you receive a *LIBRARY NOT FOUND* error message.

Note: If the result of ⎕*FNAMES*, ⎕*NNAMES*, ⎕*LIBS*, and ⎕*WSID* contains full path names, the width of the result is adjusted to accommodate them.

Printing Data

APL★PLUS II/386 provides several ways for you to print data. This section assumes that you have a compatible printer and that it is connected to your personal computer. (See the list of supported hardware in the *Installation Guide*.) Graphics support is available for some IBM graphics printers, the Hewlett-Packard LaserJet series, and for graphics printers in the Epson MX, RX, and FX series, even if you do not have a graphics adapter. (See the Using Graphics Capabilities chapter for details on printing graphics.)

You can specify which port, parallel or serial, you want to use for printing. You do this by specifying the `Printerport=` parameter in the configuration file or on the APL command line. For PostScript printers, you must set `Printerport=` to `10`.

You can specify whether you want linefeeds sent to your printer. You do this by specifying the `Linefeed=` parameter in the configuration file or on the APL command line. The default is on, which means that the system assumes that the printer will not move the paper up when it reaches the end of a line. To make the printer move the paper, the system adds an ASCII linefeed character after each newline character in the data sequence it sends. See the Displaying and Printing APL Characters chapter in the *Utilities Manual* for more information.

Keystrokes for Printing

You can enable automatic printer echo in three ways:

- Select Printer On from the Util Menu.
- Press Ctrl-PrtSc.
- Use `1 ⎕POKE 116`.

Note: If you have a PostScript printer, you must use the `printerport=10` startup parameter or execute

```
        10 ⎕POKE 125
```

to use Shift-PrtSc or Ctrl-PrtSc.

"Prt" displays in the status line unless a long workspace name overwrites it. When you turn the printer on, it prints everything you type as soon as you press Enter. In terminal mode, both sides of the data exchange are printed.

Regardless of the printer echo setting, you can print the screen or a tagged region at any time.

If you want to print the contents of your computer's screen, press Shift-PrtSc. This keystroke takes a "snapshot" of the current screen and sends it to the printer.

You can print tagged blocks or regions in APL and editing sessions. Press Alt-PrtSc to print the tagged block or region. The tagged region need not be visible on the screen or even in the current session.

Using ⎕*ARBIN* for Printing

You can also use the system function ⎕*ARBIN* to print. Use a singleton left argument to identify the printer port you want to use. The right argument is the specific array you want printed. For example,

```
      3 ⎕ARBIN 'I LIKE APL',⎕TCNL
```

sends the character array *I LIKE APL* to your printer using port 3, LPT1. (Port 10 is used for APL character support. See the next section.)

Other elements of ⎕*ARBIN* allow you to specify a translation type; for example, to separate overstrikes into their component characters. See the Communications chapter for details on ⎕*ARBIN*.

On the serial ports, ⎕*ARBIN* supports transmission protocols used by many buffered printers. These printers can receive data at higher transmission speeds than they can print. The printer continues to print until the buffer is nearly empty, while it signals the computer to stop sending data. Then, the printer signals the computer to resume transmission.

Printing APL Fonts

APL★PLUS II/386 software includes a built-in output translation that displays a complete character set on certain graphics printers, including the IBM Personal Computer graphics printer and the Epson MX, RX, and FX series printers. A separate output translation prints the traditional APL character set on printers, if the printer accepts arbitrary overstrikes of APL symbols.

Faster speed and print quality are achieved when you use a printer that accepts a downloaded font. APL fonts for a variety of printers are included with APL★PLUS II/386.

A downloaded character set is not saved when the printer power is turned off; you must load it each time you turn on your printer. The APL character set is downloaded **after** you start your session.

Workspaces for Printing

The *PRINTERS* workspace, supplied with your system, contains the functions necessary to use a downloaded character set on non-PostScript printers. To use the workspace, you must have the distributed APL component file, named *PRINTERS*, in the same directory as the workspace. To use a downloaded character set, follow these steps.

1. Before you start APL, run the appropriate terminate-and-stay-resident (TSR) program from DOS to install the port 10 driver in your system. APL★PLUS II/386 uses these TSRs to support various types of fonts:

   - APLPRINT.COM for non-PostScript printers
   - APLPS.COM for PostScript printers.

   For details, see the Displaying and Printing APL Characters chapter in the *Utilities Manual*.

2. For non-PostScript printers, start APL and load the *PRINTERS* workspace. Run *SELECT* to select a supported printer. This program downloads the character translate tables and font selectors. (The APLPS.COM program handles printer setup for PostScript printers.)

   SELECT creates several global variables for the selected printer. You also see a message that informs you of the command that reinitializes the printer should you turn it off.

3. In APL, use port 10 rather than ports 3 or 4 to direct output to your printer; for example

```
        10 ⎕ARBIN output,⎕TCNL
```

   sends *output* to your printer using the port 10 driver.

4. When you finish your APL★PLUS II/386 session, run APLPRINT /R or APLPS /R to remove the TSR from memory.

APL★PLUS II/386 also comes with three other workspaces for printing data: *LASERJET*, *EPSON*, and *POSTSCRP*.

- ***LASERJET*** **Workspace**
  The *LASERJET* workspace contains utility functions for the Hewlett-Packard LaserJet+, LaserJet II, and LaserJet III printers. You must first install the port 10 driver, then download an APL character set with functions in the *PRINTERS* workspace.

- ***EPSON*** **Workspace**
  The *EPSON* workspace has utility functions that allow you to set and test various modes for printers in the Epson FX series. Some functions require that you first install the port 10 driver, then download an APL character set with functions in the *PRINTERS* workspace.

- ***POSTSCRP*** **Workspace**
  The *POSTSCRP* workspace has utility functions for PostScript printers. You can use the functions in the *POSTSCRP* workspace to set PostScript options within APL. You must first install the Port 10 driver to use this workspace.

For more information on the functions in these workspaces, see the Displaying and Printing APL Characters chapter in the *Utilities Manual*.

2

Editing with the Session Manager

APL★PLUS II/386 combines an APL language interpreter with a user interface called the session manager. The session manager controls a "ring" of different sessions, with each session appearing in its own window. Each session allows you to interact with the APL★PLUS interpreter, edit APL objects, or run an application. You move from one session to another by using a menu, a mouse, or keystrokes.

This chapter describes the different types of sessions and explains how to use the windowing and editing features of the session manager.

- "Session-Manager Menus" explains how to use the menus to customize APL★PLUS II/386.
- "Sessions and Windows" describes the different kinds of windows, explains how to manipulate them, and explains how to create and manipulate sessions within those windows.
- "Editing Techniques" describes the editing capabilities of the session manager.

When a technique requires you to follow certain steps, those steps are numbered. When there is more than one way to perform a task, the alternative methods are shown.

Session Manager Menus

You can use pull-down menus to customize APL★PLUS II/386. The menus are normally hidden, but you can display them when you want to use them. You can display the menu bar in four ways.

- Place the mouse pointer in a blank area of the window and double click the left button.
- Place the mouse pointer in a blank area of the status line and click either button.
- Place the mouse pointer on the Menu icon in any window border and click the left button.
- Press Ctrl-/.

The first time you activate the menus, the Session Menu is highlighted. Subsequently, the menu you last used is highlighted.

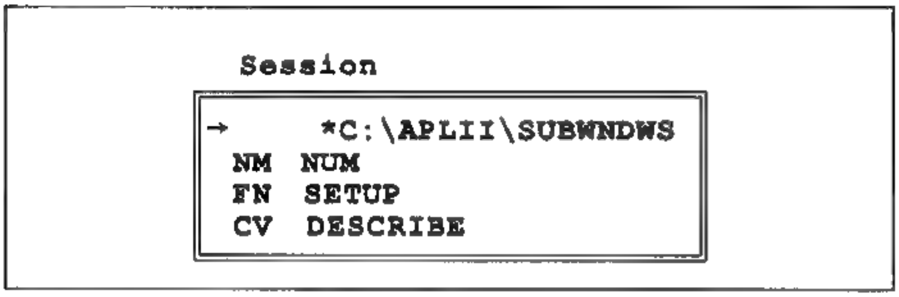
The menu bar shows these menus:

- Session
- Keyboard
- Open
- Close
- Search
- Edit
- Help

A small arrow indicates the current option on each menu. (You can also specify most of these options in the configuration file or change most of them with special keystrokes.)

The Session Menu

The Session Menu, shown in Figure 2-1, displays the current active sessions.



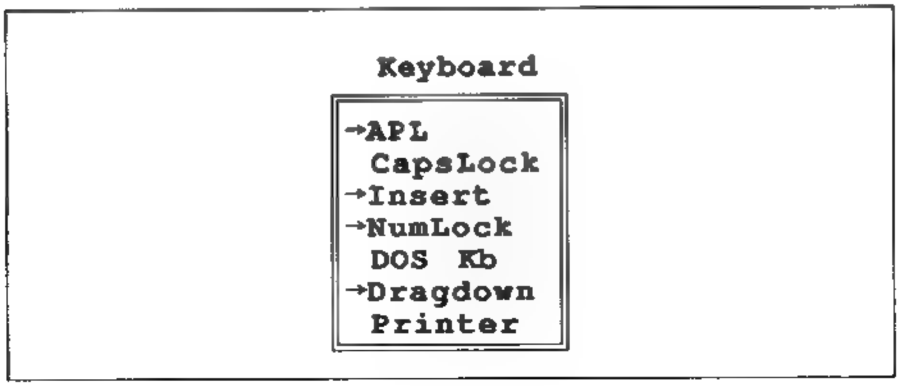
```
→     *C:\APLII\SUBWNDWS
 NM  NUM
 FN  SETUP
 CV  DESCRIBE
```

Figure 2-1. The Session Menu

The figure shows that the active sessions are a clear workspace, an editing session containing a function named *SETUP*, an editing session containing a numeric matrix named *NUM*, and an editing session containing a character vector named *DESCRIBE*.

The Keyboard Menu

The Keyboard Menu (Figure 2-2) allows you to specify your keyboard options. You have seven options.



```
Keyboard
→APL
 CapsLock
→Insert
→NumLock
 DOS  Kb
→Dragdown
 Printer
```

Figure 2-2. The Keyboard Menu

- **APL**
  Use this option to specify either the APL keyboard or the Text keyboard.
- **CapsLock**
  Use this option to specify whether you want the system to display shifted letters with the Text keyboard.
- **Insert**
  Use this option to specify whether you want to use insert mode or replace mode.
- **NumLock**
  Use this option to specify whether you want the numeric pad to display numbers or perform cursor movements.

- **DOS Kb**
  Use this option to specify that you want your keystrokes to be interpreted by DOS.

- **Dragdown**
  Use this option to specify that you want to use the dragdown feature. This feature lets you edit a line, then press Enter to copy it to the bottom of the screen and process it. The original line remains unchanged.

- **Printer**
  Use this option to activate printer echo of all output to the screen.

The Open Menu

The Open Menu (Figure 2-3) allows you to create a new session or enter terminal mode. You have seven options to choose from.

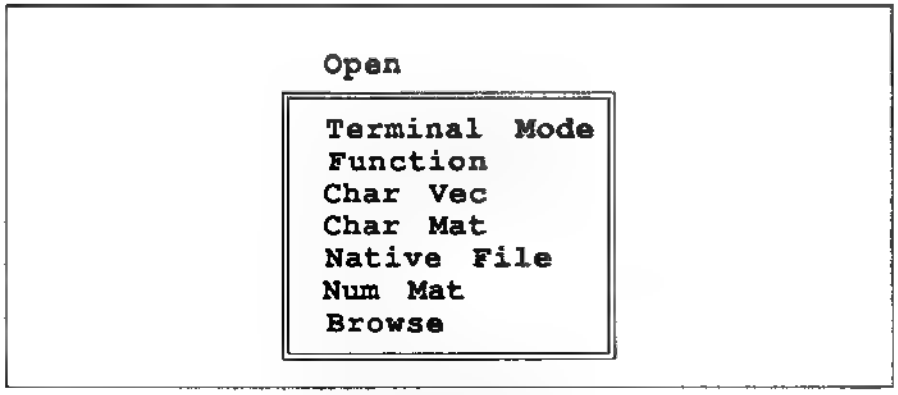


Figure 2-3. The Open Menu

- **Terminal mode**
  Terminal mode is a special communications facility that allows you to use your computer as a terminal to another computer. While not a separate session in itself, input to and output from terminal mode is recorded in the APL session. For details, see the Communications chapter.

- **Function**
  Use this option to create a function editing session.

- **Char Vec**
  Use this option to create a character vector editing session.

- **Char Mat**
  Use this option to create a character matrix editing session.

- **Native File**
  Use this option to create a native file editing session.

- **Num Mat**
  Use this option to create a numeric matrix editing session.

- **Browse**
  Use this option to view, but not change, an existing object.

The Close Menu

The Close Menu (Figure 2-4) allows you to end an editing session or to save changes you made and continue the session. This menu gives you three options.

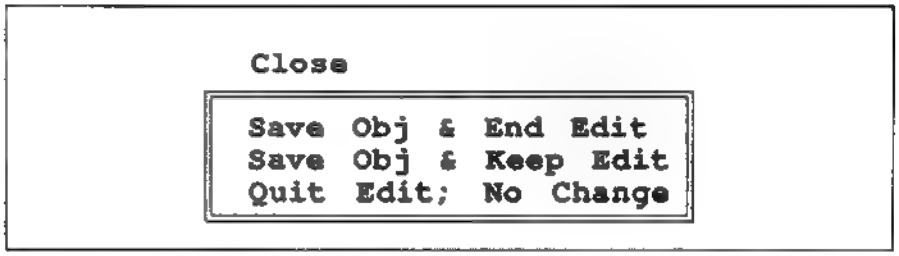


Figure 2-4. The Close Menu

- **Save Obj & End Edit**
  Use this option to save the object and end the editing session.
- **Save Obj & Keep Edit**
  Use this option to save the object and continue the editing session.
- **Quit Edit; No Change**
  Use this option to end the session without saving your changes.

The Search Menu

The Search Menu (Figure 2-5) lets you select the behavior of the search and replace commands for strings or tokens in either APL or character editing sessions. A **string** is any sequence of characters (including numbers). A **token** is an APL syntactic element, such as ⎕*FCREATE*, 4.5, or quote quad (⍞). Your search can be case sensitive. There are two options.

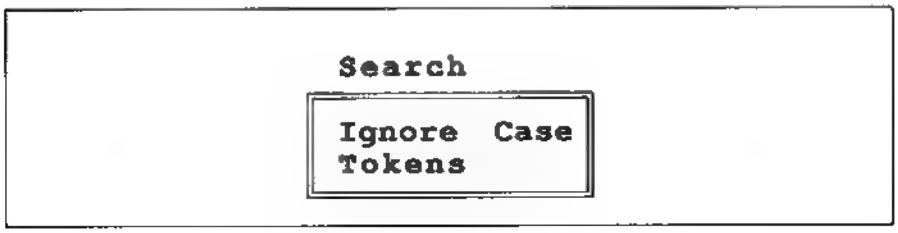


Figure 2-5. The Search Menu

- **Ignore Case**
  Use this option to search for a string of characters, regardless of case. Otherwise, the system searches for a string of characters that match the case of the characters as you entered them.

- **Tokens**
  Use this option to search for APL syntactic elements. Choosing Tokens means that the search will not find a name if it happens to be part of a longer character string. Token searches are case sensitive.

Note: When editing a numeric matrix, you can only search for entire numbers .

The Edit Menu

The Edit Menu (Figure 2-6) lets you modify properties of objects in the full-screen editor and the characteristics of the window. There are 10 options.

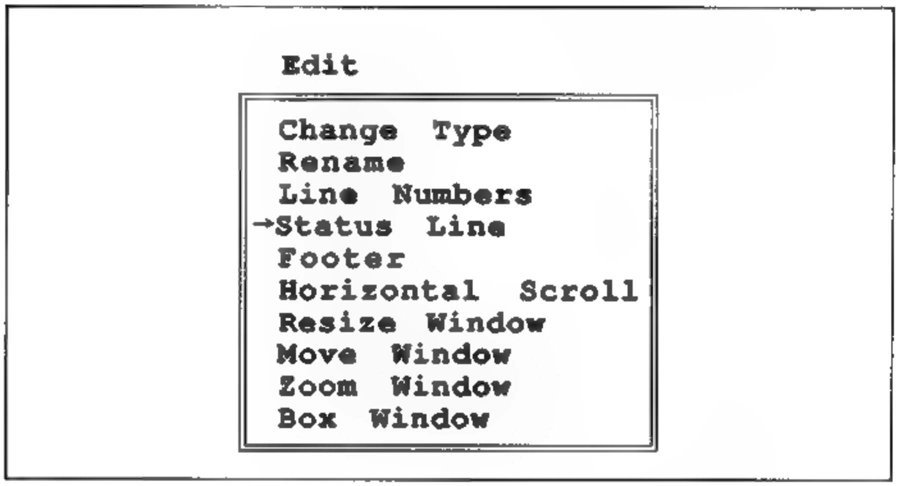


Figure 2-6. The Edit Menu

- **Change Type**
  Use this option to change the type of a character object; for example, a character vector to a character matrix.

- **Rename**
  Use this option to change the name of an object.

- **Line Numbers**
  Use this option to turn the line numbers on or off when you edit a character object. The `editnums=` startup parameter determines the default state. You cannot turn line numbers off in numeric editing sessions.

- **Status Line**
  Use this option to turn the display of the status line on or off in the current session.
- **Footer**
  Use this option to replace the status line with a footer, which contains the name of the current session centered in the bottom line.
- **Horizontal Scroll**
  Use this option to specify whether you want to use horizontal scrolling in the session.
- **Resize Window**
  Use this option to resize the current window.
- **Move Window**
  Use this option to move the current window elsewhere on the screen.
- **Zoom Window**
  Use this option to enlarge the current window to full screen or shrink a full-screen window back to a previously used smaller size.
- **Box Window**
  Use this option to create or remove a box around the current window.

The Help Menu

The Help Menu (Figure 2-7) is a ready source of on-line help. It has two options.

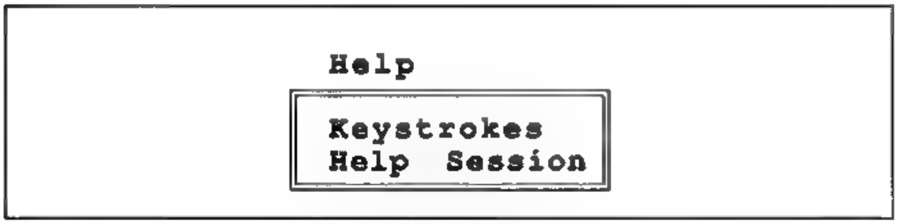


Figure 2-7. The Help Menu

- **Keystrokes**
  Use this option to display a table showing all of the keystroke combinations allowable in the sessions.
- **Help Session**
  Use this option to start the built-in Help facility.

For other ways of getting help, see the Getting Started chapter.

Choosing Menus

You can select menus from the menu bar in two ways: with the mouse or with keystrokes.

- To select menus with the mouse, move the mouse pointer to the menu title you want to see and click the left button. To exit the menus, move the pointer anywhere off any menu title and click the left button.
- To select menus with keystrokes, use the right and left arrows (→ and ←), Tab, and Shift-Tab.

When you use the arrow keys to move across the menu bar, the selections for that menu appear. The selections disappear when you move the cursor off that menu.

When you use Tab and Shift-Tab to move across the menu bar, the menu selections do not display. When you reach the menu you want, press Enter or Down Arrow (↓) to display the selections for that menu.

Selecting Options from Menus

You can use the mouse or keystrokes to select menu options.

- **With the Mouse**
  - ❑ To select one item at a time, move the mouse pointer to the selection you want and click the left button. Once you select an option, the session manager toggles any on/off switches and the menus disappear.
  - ❑ To select more than one item at a time, move the mouse pointer to the selection you want, press Ctrl, and click the left button. The menus do not disappear, so you can continue to make selections.
  - ❑ Place the mouse pointer on the menu title you want. Hold down the left button. Move to the choice you want to select or toggle and release the button.
  - ❑ To turn the menus off without making a selection, move the mouse pointer anywhere outside the menu or menu bar, and click the left button.
- **With Keystrokes**
  You use six keys and the Spacebar to select options from a menu: the up and down arrows (↑ and ↓), Enter, Insert (Ins), Delete (Del), and Escape (Esc).
  - ❑ Press the up and down arrow keys to highlight your choice, then press Enter to make the selection and close the menu.

- Press Ins to choose more than one selection from a menu. When you finish choosing, press Esc to leave the menus.
- Press Del to turn off a selection. **Note:** If you turn off a selection with Del and then press Enter, you turn the selection on again. Press Esc to close the menus.
- Press the Spacebar to toggle a selection.

Sessions and Windows

Each session in APL★PLUS II/386 runs in its own window. There are three general classes of windows: APL, editing, and application.

- The APL window holds the APL session, which is a record of the input you entered and the output produced. You can edit this record. When you start APL★PLUS II/386, you are initially in the APL session window. The size and location of the APL session window is given by the system variable ⎕*WINDOW*.
- Editing windows allows you to run editing or browse sessions. In editing sessions, you can edit APL functions, character or numeric arrays, and native files. In browse sessions, you can only view, but not change, APL functions, character or numeric arrays, and native files.
- Application windows contain user interactions that can run separate from the APL window or edit windows. You must supply and format the data for the window, define or program all user interactions, and establish exit methods. See the Advanced Techniques chapter for more information on creating and using application windows.

You can have up to 32 windows open simultaneously. Generally, the window with input focus from the session manager is the only window active at any one time.

Windows can be surrounded by a box or without a box (the default). Boxed windows also have icons that allow you to resize, scroll, and move them easily. Figure 2-8 shows a boxed window.

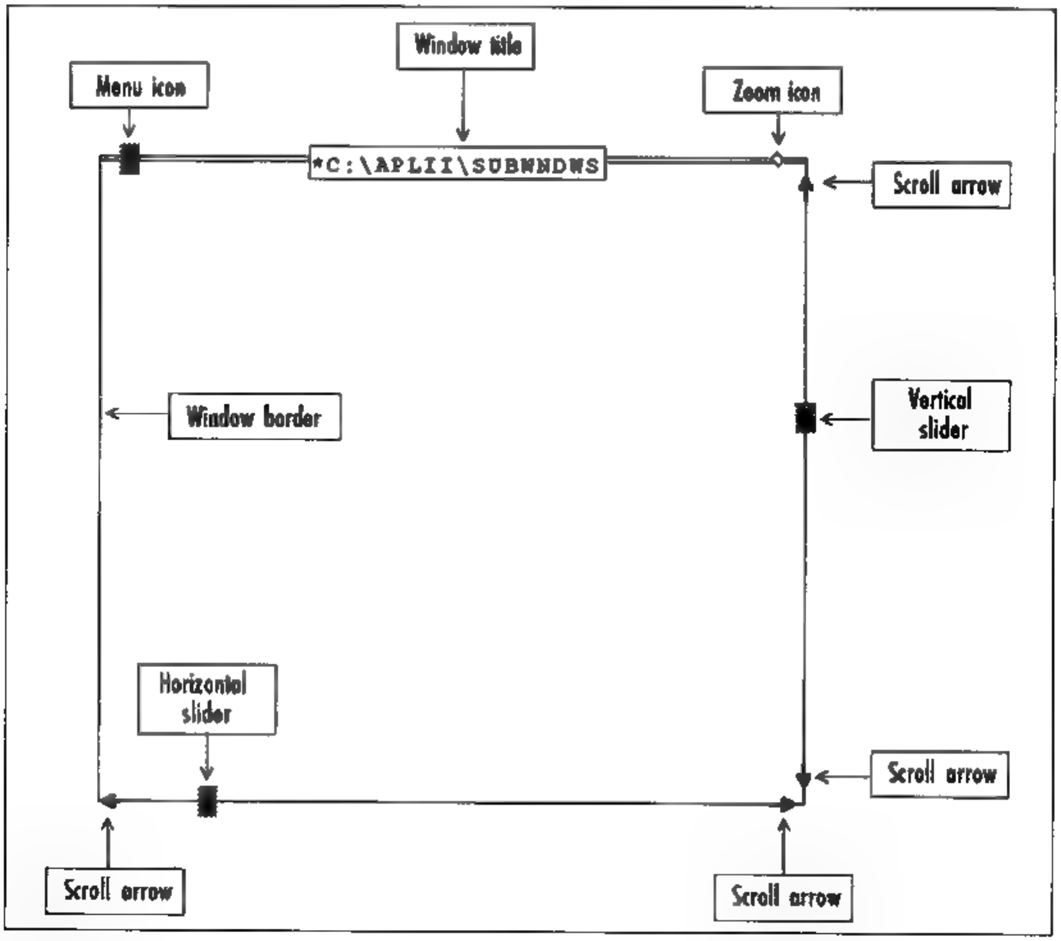


Figure 2-8. Boxed Window

Press Ctrl-Shift-B to toggle between boxed and unboxed windows. The `wstate` startup parameter and the *⎕WSTATE* system variable allow you to change the default from unboxed windows to boxed windows. To start APL★PLUS II/386 with boxed windows, put the line

```
    wstate 29=1
```

in the configuration file or use it on the command line when you start APL★PLUS II/386. You can use *⎕ECTRL* to change the appearance of an existing window. See the Advanced Techniques chapter for more information.

Manipulating Windows

You can manipulate windows on the screen in many ways. These techniques allow you to arrange the windows on the screen for maximum usability.

- You can resize windows from full-screen size to any size you want. The minimum width is 10 columns; the minimum height depends on whether the window is boxed and has a status line.
- If you resize a window to a size smaller than the screen, you can "zoom" the window back to full-screen size and shrink it back again.
- You can move any window in the ring to any location on the screen. Except for the APL window, you can position a window partially off the screen.
- You can bring a window to the front of the edit ring.
- You can scroll the information in a window, both vertically and horizontally.

Resizing and Zooming Windows

Resizing windows allows you to see several windows on the screen at the same time. The active window may overlap other windows, depending on the size and number of windows on the screen.

- **With the Mouse, Boxed Windows**

  1. Place the mouse pointer on any corner of the window.
  2. Press the left button and drag the window to its new size.
  3. Release the button when the size is right.

- **With the Mouse, Using Menus**

  1. Use the mouse to select "Resize Window" from the Edit Menu.

  2. Place the mouse pointer on an edge of the screen.

  3. Press the left button and drag the mouse to resize the window. Drag the top or bottom edge to resize the window vertically; drag to the left or right edge to resize the window horizontally. Drag to a corner to resize the window both vertically and horizontally.

  4. Release the button when the size is right.

- **With Keystrokes**

  1. Press Ctrl-Shift-A or use Enter to select "Resize Window" from the Edit Menu. The window highlight changes, a block cursor appears in the upper-left corner, and a message appears in the status line.

  2. Press Home, End, PgDn, or PgUp to move the block cursor to the edge you want to start the resizing operation from.

  3. Use the cursor keys to change the window size. Changes occur relative to the top and left side of the window.

  4. Press Enter when the size is right.

Once you resize a window to less than full-screen size, you can enlarge (or "zoom") it to full-screen size again, and shrink it back to the smaller size. You can zoom windows with the menus, the mouse, or keystrokes.

- **With the Mouse**
  If the window is boxed, click the zoom icon (see Figure 2-8). The zoom icon changes to the shrink icon. To shrink the window to its previous size, click the shrink icon.

- **With Keystrokes**
  Press Ctrl-Shift-Z. Press Ctrl-Shift-Z again to shrink the window to its previous size.

- **With Menus**
  Select "Zoom Window" from the Edit Menu. Select "Zoom Window" again to shrink the window to its previous size.

Moving Windows

- **With the Mouse**

  1. Use the mouse to select "Move Window" from the Edit Menu.
  2. Move the mouse pointer to an edge of the screen.
  3. Press the left button and drag the window to its new position.
  4. Release the button to place the window in the new position.

- **With the Mouse, Boxed Window**

  1. Place the mouse pointer anywhere in the title bar.
  2. Press the left button and drag the window to its new position.
  3. Release the button to place the window in the new position.

- **With Keystrokes**

  1. Press Ctrl-Shift-M or use Enter to select "Move Window" from the Edit Menu. The window highlight changes and a message appears in the status line.

  2. Use the cursor keys to move the window to its new location.

  3. Press Enter to complete the move.

Note: You cannot move any window completely off the screen. You cannot move any part of the APL window off the screen.

Scrolling in Windows

You can scroll the data in windows either vertically or horizontally. You can scroll windows vertically with the mouse or with keystrokes.

- **With the Mouse**
  - ❑ Place the pointer on the row number in the status line. To scroll up, press the left button. The number decreases as the window scrolls. To scroll down, press the right button. The number increases as the window scrolls.

  - ❑ If the window is boxed, place the pointer on the vertical scroll bar on the right border. Press the left button and drag the scroll bar up or down. The window scrolls as you drag the scroll bar.

  - ❑ If the window is boxed, place the pointer on the up or down scroll arrows on the right border. Click the left button to scroll one line in the direction of the arrow.

  - ❑ If the window is boxed, click on the right border between the slider and the scroll arrow.

- **With Keystrokes**
  - ❑ If the cursor is at the top or bottom of the screen, press the up and down arrow keys to scroll one line in the direction of the arrow.
  - ❑ Press PgUp or PgDn to scroll the window one line up or down without moving the cursor.
  - ❑ Press Ctrl-PgUp or Ctrl-PgDn to scroll the window one window full of lines without moving the cursor.

To scroll horizontally, you must first turn horizontal scrolling on (it is off by default). You can turn on horizontal scrolling using the menus or keystrokes.

- **With Keystrokes**
  Press Ctrl-Shift-H.
- **With Menus**
  Select Horizontal Scroll from the Edit Menu.

You can also use the `wstate=31` startup parameter to turn on horizontal scrolling in the APL session at startup, or change element 31 of *⎕WSTATE* to control horizontal scrolling when editing sessions are created.

You can use the mouse or keystrokes to scroll horizontally.

- **With the Mouse**
  - ❑ If the window is boxed, place the pointer on the horizontal scroll bar in the bottom border. Press the left button and drag the scroll bar right or left. The window scrolls as you drag the scroll bar.
  - ❑ If the window is boxed, place the pointer on the left or right scroll arrows on the bottom border. Click the left button to scroll one column in the direction of the arrow.
  - ❑ If the window is not boxed, place the pointer at the left or right edge of the window. Press the right button and drag the pointer in the direction you want to scroll.

- ❑ If the window is boxed, click on the bottom border between the slider and the scroll arrow.

- **With Keystrokes**
  - ❑ Use the left and right arrow keys to scroll horizontally. Move the cursor to the edge of the window, then press the keys to scroll in the direction of the arrow.
  - ❑ Alt-Tab and Alt-Shift-Tab move the screen horizontally one position without moving the cursor. Ctrl-Tab and Ctrl-Shift-Tab move the screen horizontally one screen width.

Working with Sessions

This section explains basic session manipulation. You can use the mouse, the menus, or keystrokes to perform your work.

Creating New Sessions

APL★PLUS II/386 starts up in an APL session. You can start in a specific workspace or in a clear workspace depending on how you start the system.

Editing Sessions

You can create a new editing session with the mouse, with keystrokes, with the menus, or with system commands and system functions.

- **With the Mouse**
  - ❑ Place the mouse pointer on an object name and double-click the left button.
  - ❑ To edit a native file, place the mouse pointer on the period that separates the file extension from the file name.
- **With Keystrokes**
  - ❑ **Ctrl-Shift-I**
    Move the cursor to the name of the object you want to edit. To edit a native file, place the cursor on the period that separates the file extension from the file name, then press Ctrl-Shift-I. This technique is an easy way to begin an editing session without typing the name of the object. For example, if you are editing a function, you can examine a subroutine by placing the cursor on the name of the subroutine and pressing Ctrl-Shift-I. If the cursor is not

directly over a name, the system searches that line, first right, then left, for the nearest name.

- ❑ **Ctrl-I**
  Press Ctrl-I. The status line prompts you for the object name. Type the object name you want to create and press Enter. The system uses the value of the `edittype=` startup parameter to determine what type of object you want to edit; the default is a character vector.

  — To create a function editing session, precede the object name with a del (∇) character.

  — To create a numeric matrix editing session, precede the object name with a pound sign (#) character.

  — To create a native file editing session, include the period that separates the file extension from the file name.

  — To change the type of a non-numeric object after entering the editor, press Ctrl-A or use the Edit Menu.

■ **With Menus**
Display the Open Menu and select the type of session you want. If you select an editing session, the status line prompts you for the object name. Type the name and press Enter.

■ **With System Commands or System Functions**
Use the system command *)EDIT* or the system function *⎕EDIT*. Enter *)EDIT objectname* or *⎕EDIT 'objectname'* to create a new editing session. (For details on *)EDIT*, see the System Commands chapter in the *Reference Manual*; for details on *⎕EDIT*, see the System Functions, Variables, and Constants chapter in the *Reference Manual*. You can also use *⎕ED* under program control. For information on this technique, see the Advanced Techniques chapter later in this manual.)

Browse Sessions

A browse session lets you examine an object or file without being able to change it, and without bringing the entire item into the session manager memory. This is useful when you want to ensure that you do not inadvertantly change an item, or when the object is too large for the amount of memory available in the session manager. There are several visible differences between a browse session and an editing session.

- Pressing a key that would ordinarily change the contents of an item — such as a character key — results in a beep rather than a change. Ctrl-H displays a list of the keystrokes available in a browse session.
- Horizontal scrolling is always in effect.
- Browse sessions are listed in the Session Menu with their type indicators in lowercase; for example, if you browse a native file called WIZBANG.SF, "nf WIZBANG.SF" appears in the Session Menu.
- You can copy lines from a browse session to an editing session using Ctrl-A. The system copies any tagged lines into a new editing session. If you do not tag any lines, the system copies only the lines displayed on the screen.

You can start a browse session with keystrokes, with the menus, or with ⎕*EDIT*. The item you want to browse must exist, otherwise the system displays an error.

- **With Keystrokes**
  Press Ctrl-Shift-W. The status line prompts you for the object name. Type the object name you want to create and press Enter.
- **With Menus**
  Display the Open Menu and select Browse. The status line prompts you for the object name. Type the name and press Enter.

- **With System Functions**
  Use this variant of ⎕*EDIT* to create a browse session:

  ```
  ⎕EDIT 0 ⎕EDIT 'objectname'
  ```

  You can also use ⎕*ED* under program control. For information on this technique, see the Advanced Techniques chapter later in this manual.

You can change a browse session to an editing session any time by pressing Ctrl-A. A message on the status line prompts you for the object name. You can type a new name or use the same name. When you convert a browse session to an editing session, only the lines in the current window become part of the editing session. However, if you have tagged lines, those lines become part of the editing session.

Moving through Sessions in the Ring

You can move around the nonapplication sessions in the ring with the mouse, with keystrokes, with menus, or with the system commands and system functions.

- **With the Mouse**
  - If you have more than one window displayed on the screen, move the mouse pointer to the window you want and click the left button.
  - If you are using the default status line, place the pointer in the status line on the name of the session. Click the left button to move to the previous session in the ring; click the right button to move to the following session in the ring.
- **With Keystrokes**
  - Press Ctrl-Shift-P to move to the previous session in the ring; press Ctrl-Shift-N to move to the next session in the ring.

- ❑ Press Ctrl-X to return to the APL session from any editing session in the ring, without ending the edit session. If you press Ctrl-X in the APL session, you return to the last session you left using Ctrl-X.

- ■ **With Menus**
  Display the Session Menu and select the session you want.

- ■ **With System Commands or System Functions**
  Enter one of the following

```
)EDIT objectname
⎕EDIT 'objectname'
⎕EDIT  'windownumber'
```

  from the APL session to move to an existing session in the ring. The last variant moves to browse sessions.

Renaming Sessions

You can rename an editing session if you are not editing a function (rename a function by editing its header). Follow these steps.

■ **With Keystrokes**

1. Press Ctrl-Shift-R. A message appears in the status line prompting you for the new name.
2. Type the new name and press Enter.

■ **With Menus**

1. Display the Edit Menu and select Rename.
2. Type the new name and press Enter.

Saving and Ending Sessions

You can end a session using keystrokes or the menus.

- **With Keystrokes**
  - ❑ Press Ctrl-E to save an editing session to a workspace object and end the session.
  - ❑ Press Ctrl-Q to end any browse session, or to end an editing session without saving the editing you have done.
- **With Menus**
  Display the Close Menu and choose the option you want.

Regardless of how you end a session, the system always moves back to the previous session in the ring.

You can also save any changes and continue your session. To save an editing session as a workspace object and continue the session, press Ctrl-S.

Types of Editing Sessions

The session manager allows you to edit the APL session, APL functions, APL objects, and native files. You can have up to 32 simultaneous sessions, including the APL session.

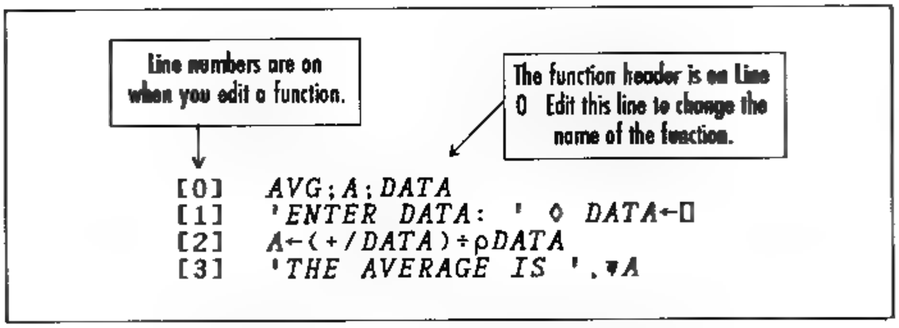
There are two types of editing sessions: character editing sessions and numeric editing sessions.

Character Editing Sessions

Character editing sessions are those in which you edit functions or objects containing character data. Each single character, whether a letter in a word or a digit in a number, can be individually edited. There are four types of character editing sessions:

- character matrix
- character vector
- function
- native file

A sample function editing session is shown in Figure 2-9.


Line numbers are on when you edit a function.
The function header is on Line 0 Edit this line to change the name of the function.

```
[0]   AVG;A;DATA
[1]   'ENTER DATA: ' ⋄ DATA←⎕
[2]   A←(+/DATA)÷ρDATA
[3]   'THE AVERAGE IS ',⍕A
```

Figure 2-9. Function Editing Session

Table 2-1 shows the cursor movement keystrokes available in character editing sessions.

Table 2-1. Cursor and Window Movement — Character Editing Session

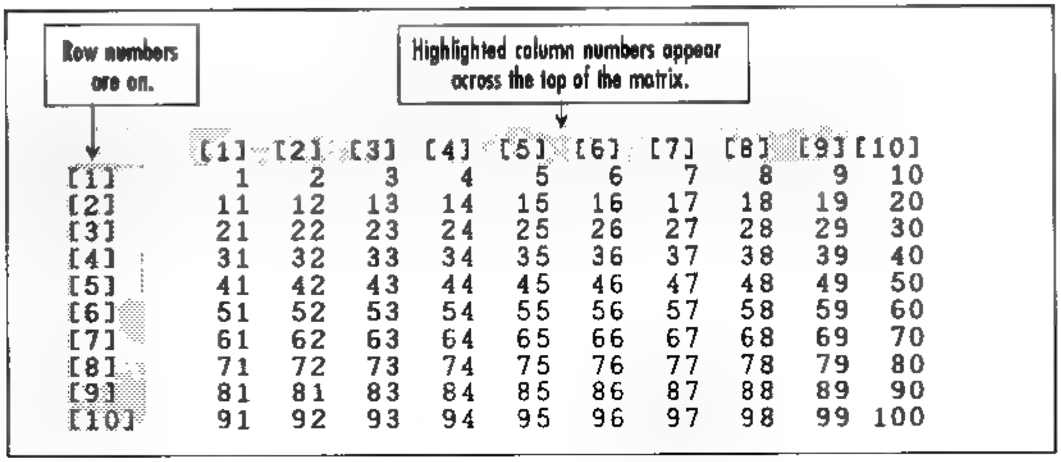
Keystroke	Description
↑ ↓ ← →	Move one row or position in indicated direction.
Alt-↓	Move to current column in last line.
Alt-↑	Move to current column in first line.
Alt-←	Move to left edge of current line.
Alt-→	Move to end of current line.
Alt-End	Move to beginning of last line.
Alt-Home	Move to beginning of first line.
Alt-PgDn	Move to top line of clear screen at bottom of session.
Alt-Tab	Move window right one column.
Alt-Shift-Tab	Move window left one column.
Ctrl-↓	Move four lines down.
Ctrl-↑	Move four lines up.
Ctrl-←	Move eight positions left.
Ctrl-→	Move eight positions right.
Ctrl-J	Move forward to specified row in column.
Ctrl-Shift-J	Move to specified column in current row.
Ctrl-End	Move to bottom left.
Ctrl-Home	Move to top left.
Ctrl-PgDn	Move window down one full window, less one row.
Ctrl-PgUp	Move window up one full window, less one row.
Ctrl-Tab	Move window right one full window.
Ctrl-Shift-Tab	Move window left one full window.
End	Move one position past end of current line.
Home	Move to first non-blank character on line.
PgDn	Move window down one line.
PgUp	Move window up one line.
Tab/Shift-Tab	Move one tab stop right or left.
Shift-Esc *n*-PgUp	Move up *n* lines.
Shift-Esc *n*-PgDn	Move down *n* lines.
Shift-Esc *n*-↑	Move up *n* rows.
Shift-Esc *n*-↓	Move down *n* rows.
Shift-Esc *n*-→	Move right *n* positions.
Shift-Esc *n*-←	Move left *n* positions.

Numeric Editing Sessions

Numeric editing sessions are those in which you edit numeric data objects. Each single number, rather than each digit in a number, is an item that you can edit. In this version of APL★PLUS II/386, you can edit homogeneous, unnested numeric vectors and matrices. Numeric arrays can contain digits, decimal points, negative signs, or an *E* to indicate exponential notation. Row and column numbers can begin at 1 or at 0, depending on the value of ⎕*IO*.

Note: The rank of a numeric array can be reduced to a vector if it has only one row or to a scalar if it has only one element, but it is not reduced to a rank less than that of the original array.

Figure 2-10 shows a sample numeric editing session.


Row numbers are on.
Highlighted column numbers appear across the top of the matrix.

```
        [1] [2] [3] [4] [5] [6] [7] [8] [9][10]
[1]       1   2   3   4   5   6   7   8   9  10
[2]      11  12  13  14  15  16  17  18  19  20
[3]      21  22  23  24  25  26  27  28  29  30
[4]      31  32  33  34  35  36  37  38  39  40
[5]      41  42  43  44  45  46  47  48  49  50
[6]      51  52  53  54  55  56  57  58  59  60
[7]      61  62  63  64  65  66  67  68  69  70
[8]      71  72  73  74  75  76  77  78  79  80
[9]      81  81  83  84  85  86  87  88  89  90
[10]     91  92  93  94  95  96  97  98  99 100
```

Figure 2-10. Numeric Editing Session

Warning: APL★PLUS II/386 places a character image of the numeric array in a numeric editing session. Only the digits you see are used to build numeric variables when you end a session, save changes in a session, or write the array to a variable. If you require levels of precision beyond that provided by the current value of ⎕*PP* (default = 10), set ⎕*PP*←17 *before* you begin a numeric editing session.

Cursor movement in a numeric editing session differs from cursor movement in the other sessions. The cursor appears as a "highlight bar," which moves from value to value in the matrix. Table 2-2 shows the cursor movement keystrokes available in numeric editing sessions.

Table 2-2. Cursor and Window Movement — Numeric Editing Session

Keystroke	Description
↑ ↓ ← →	Move one row or column in indicated direction.
Alt-↓	Move to bottom of matrix.
Alt-↑	Move to top of matrix.
Alt-←	Move to left edge of matrix.
Alt-→	Move to right edge of matrix.
Alt-End	Move to lower-right corner of matrix.
Alt-Enter	Move to first row of next column, adding a column if at edge of matrix.
Alt-Home	Move to upper-left corner of matrix.
Alt-Tab	Move window right one column.
Alt-Shift-Tab	Move window left one column.
Ctrl-↓	Move down one full window.
Ctrl-↑	Move up one full window.
Ctrl-←	Move left one full window.
Ctrl-→	Move right one full window.
Ctrl-;	Toggle behavior of Tab and Shift-Tab between horizontal (default) and vertical movement.
Ctrl-End	Move to lower-right corner of window.
Ctrl-Enter	Move to first column of next row, adding a row if at edge of matrix.
Ctrl-Home	Move to upper-left corner of window.
Ctrl-J	Move to specified row in current column.
Ctrl-Shift-J	Move to specified column in current row.
Ctrl-PgDn	Move window down one full screen.
Ctrl-PgUp	Move window up one full screen.
Ctrl-Tab	Move window right one full screen.
Ctrl-Shift-Tab	Move window left one full screen.

Table 2-2. Continued

Keystroke	Description
End	Move to end of current row of window.
Home	Move to beginning of current row of window.
PgDn	Move window down one row.
PgUp	Move window up one row.
Tab	Move right or down one value; move to next row or column if at the end of the current one.
Shift-Tab	Move left or up one value; move to previous row or column if at the end of the current one.

Editing Techniques

This section describes the various editing capabilities of the session manager, including:

- searching and replacing text and numbers
- tagging blocks of text and numbers
- copying blocks of text and numbers
- moving blocks of text
- changing the structure of numeric arrays.

Bringing Objects into the Editor

The session manager allows you to bring (or "fetch") an APL object from the active workspace or bring an entire native file into an editing session so that you do not have to retype it.

■ To bring an object into an editing session, follow these steps.

1. Go to or create the editing session.
2. Press Ctrl-F. A message on the status line prompts you for the object name.
3. Type the object name and press Enter. If you are bringing in a native file and the filename has no extension, use a trailing period to distinguish the file from an APL object.

In a character session, the session manager inserts the object at the cursor, where you can then edit it.

You can only bring numeric objects into a numeric editing session. The session manager places the upper-left corner at the cursor. If there is not enough room, the system beeps.

Note: In numeric sessions, the object you bring in overwrites existing data.

Changing Object Type

Once you begin a character editing session, you can change the type of the object you are editing. If you create a new session from the Open Menu, you can select the object type then. If you create a new session by pressing Ctrl-I, the system uses the value of the `edittype=` startup parameter to determine the default session type.

You can use keystrokes or the menus to change object type.

- To change the object type with keystrokes, follow these steps.

  1. Press Ctrl-A to change the object type. A message on the status line prompts you for the new object type:

F	Function
M	Character matrix
V	Character vector
N	Native file

  2. Type the appropriate letter and press Enter. The new object type is reflected in the status line.

- To change the object type with the menus, select the "Change Type" option from the Edit Menu.

You cannot change the type of session to a numeric matrix, and you cannot change a numeric matrix to another type of session.

Searching for Strings and Tokens

You can search for strings or tokens in any editing session. The search can include more than one APL token. Several keystrokes allow you to search forward and backward.

- Press Ctrl-L to search forward through the session for the string, token, or number. A message on the status line prompts you for the search text.

- Press Ctrl-Shift-L to search backward through the session for the specified string, token, or number.
- Press Ctrl-N to search through the session for the next occurrence of the specified string, token, or number.
- Press Ctrl-P to search through the session for the previous occurrence of the specified string, token, or number.

The session manager highlights the string, token, or number when it finds them. In character editing sessions, you can press Ctrl-~ to toggle the case-sensitive search; press Ctrl-_ to toggle between strings and tokens. If you begin to enter the search text and press Ctrl-L or Ctrl-Shift-L again, you can toggle between strings and tokens. When using Ctrl-L, you can press Alt-F10 or the Up Arrow (↑) at the prompt to recall the most recently used search string.

Replacing Text and Numbers

In addition to searching for strings or tokens, you can replace text or numbers in your editing session.

- **Replacing Text**
  - ❑ Use replace mode to overwrite the text you want to replace. To activate replace mode, press Ins. The status line displays "Rpl." To replace text, move the cursor to the text you want to replace and type the correct text.
  - ❑ Press Del or Backspace to correct any errors you make while typing.
- **Replacing Numeric Data**
  - ❑ To replace a single number, follow these steps.
    1. Place the cursor on the number you want to replace.
    2. Type the new number and press Enter. The replacement number appears on the status line as you type.

If you make a mistake while typing the new number, use Backspace to delete the incorrect number, type the correction, and press Enter.

- ❑ To change one or more digits in a number without retyping the entire number, follow these steps.

  1. Place the cursor on the number you want to change and press Enter. The number appears in the status line.
  2. Use the left and right arrows, Home, End, Del, and Backspace to edit the number in the status line.
  3. Press Enter to replace the old value with the new value. Press Esc to stop editing the value without changing it.

If you change a value and then decide you made a mistake, leave the cursor on the incorrect number. Press Ctrl-Backspace to restore the former value.

■ **Replacing Text and Numbers Automatically**
To replace text and numbers automatically, follow these steps.

1. Press Ctrl-R to replace text or numbers in the session. A message on the status line prompts you for the string or number to be replaced. If you press Alt-F10 or the Up Arrow (↑) in response to the prompt, the system supplies the most recently used search string.
2. Type the string and press Enter. A message on the status line prompts you for the replacement string or number. If you press Alt-F10 or the Up Arrow (↑) in response to the prompt for a replacement string, the system supplies the most recently used search string.
3. Type that string or number and press Enter.
4. The system replaces the first occurrence of the target. A message on the status line asks whether you want to replace the next occurrence, undo this change, replace all occurrences, or quit.

5. Type N to replace the next occurrence, U to undo the change you just made, A to replace all occurrences, or Q to quit.

To resume the replacement after you stop, press Ctrl-4. The session manager continues the replacement from the point where you stopped.

Manipulating Text and Numbers

This section explains how to manipulate text and numbers in your sessions. Most character-editing keystrokes apply to editing sessions and to the APL session; those that are specific to either type of session are indicated.

Tagging Text and Numbers

Before you can copy, move, or erase text or numbers, you must **tag** them. Tagging tells the system what portion of the text or numeric matrix you want to work with. Tagged areas are called **blocks**.

There are three kinds of tagging.

- Line tagging means that you tag entire lines. When tagging entire lines, the tagged area begins at the start of the first line tagged, including the carriage return, and goes to the end of the last line tagged.
- Text tagging means that you tag parts of a line or line. Blocks of tagged text do not have to be rectangular.
- Rectangular tagging means that you tag a rectangular area of the session. The block can start or end anywhere on a line, but the block is always rectangular in shape.

You can have only one tagged block throughout all of your editing sessions. If you have a tagged block in one session and move to another session, you have to remove the tagging before you can tag another block. The notation *T*: in the status line indicates a tagged block.

If you are tagging with the mouse and move the mouse beyond the edge of the current window, the text scrolls in the appropriate direction. If horizontal scrolling is on, the text scrolls horizontally as well as vertically. If the edge of the window is also the edge of the screen, you can push the mouse against the edge of the window to scroll.

If you decide that you want to tag more text while you have an active tagged block, press Ctrl-+. The cursor moves to one end of the tagged region and waits for you to complete tagging.

To turn tagging off, press Ctrl-T or Ctrl-Shift-T when a block is tagged.

In numeric sessions, you can fill a tagged block with another value in the array. After tagging the block, move the cursor to the value, then press Ctrl-Shift-C or Ctrl-K. The values in the tagged block will change to the value stored at the cursor position.

Line Tagging in Character Sessions

With the Mouse

1. Place the pointer on the first line that you want to tag.
2. Press and hold down the right button.
3. Drag the pointer over the rest of the lines that you want to tag.
4. Release the button to complete the tag.

❑ With Keystrokes

1. Press Ctrl-T. A highlight indicates the starting line for the block and the status line displays the message:

   ```
   Waiting for tag
   ```

   This message is your signal to begin tagging.

2. Move the cursor up or down through the lines you want to tag.

3. When you finish tagging, press Ctrl-T again. The tagged lines assume a new highlight.

■ Text Tagging in Character Sessions

❑ With the Mouse

1. Place the mouse pointer at the beginning of the text.

2. Press Ctrl and the right button.

3. Drag the pointer to the end of the text that you want to tag.

4. Release Ctrl and the right button to complete the tag.

❑ With Keystrokes

1. Move the cursor to the beginning of the text.

2. Press Ctrl-Shift-T. The character at the cursor is highlighted. The status line displays the message:

   ```
   Waiting for text tag
   ```

3. Move the cursor to the last character of text that you want to tag.

4. Press Enter, Ctrl-T, or Ctrl-Shift-T again.

- **Rectangular Tagging in Character Sessions**
  - **With the Mouse**
    1. Place the mouse pointer at the beginning of the text.
    2. Press Alt and the right button.
    3. Drag the pointer to the end of the block that you want to tag.
    4. Release Alt and the right button to complete the tag.
  - **With Keystrokes**
    1. Move the cursor to the beginning of the text.
    2. Press Alt-Ctrl-T. The character at the cursor is highlighted. The status line displays the message:

       ```
       Waiting for rectangular tag
       ```
    3. Move the cursor to the last character that you want to tag.
    4. Press Enter to complete the tag.
- **Tagging Numbers in Numeric Sessions**
  You can tag a rectangular block of numbers in a numeric matrix with the mouse or with keystrokes. This is the only type of tagging available for numeric sessions.
  - **With the Mouse**
    1. Place the mouse pointer at one corner of the block that you want to tag and press the right button.
    2. Hold the button down and move the pointer to the opposite corner of the block.
    3. Release the button to complete the tag.

- ❑ **With Keystrokes**

  1. Press Alt-Ctrl-T at one corner of the block.

  2. Move the cursor to the opposite corner of the block. As you move the cursor, the tagged numbers are highlighted.

  3. When you reach the opposite corner of the block, press Ctrl-T or Alt-Ctrl-T again.

Copying and Moving Blocks

Once you have a block tagged, you can copy it, move it, or delete it. You can use the mouse or keystrokes to copy, move, or delete blocks.

- ■ **With the Mouse**
  - ❑ Move the pointer to the new location and press the left button. The cursor moves to the pointer and a pop-up menu appears. This menu has four options: copy, move, delete, and cancel.
  - ❑ To select one of the options from the menu, move the pointer to it and click the left button, or press the appropriate keystroke.
  - ❑ To close the menu, click the right button or choose Cancel from the menu.
- ■ **With Keystrokes**
  - ❑ To copy a character block to another location, including a different session, move the cursor to the new location and press Ctrl-C. By default, the block is inserted starting at the cursor.

- To copy a rectangular character block to another location and overwrite and existing rectangular block, ensure that

  ```
          0 ⎕POKE 945
  ```

  is set. Then press Ctrl-C to copy the block.

- To copy a numeric block to another location, move the cursor to the the upper-left corner of the new position and press Ctrl-C. You cannot copy numeric blocks to different sessions. **Note:** The block overwrites existing data.

- To move a character block to another location, including a different session, move the cursor to the new location and press Ctrl-M.

- To delete a character block, press Ctrl-D. The remaining lines are closed up and renumbered. If you delete a block by mistake, press Ctrl-Shift-U to undo the deletion.

You can also store a character or numeric block in a variable. Follow these steps.

1. Press Ctrl-V. A message on the status line prompts you for the variable name.

2. Type the variable name and press Enter. If the variable already exists, a message on the status line asks if you want to replace it. Type Y for Yes or N for No.

Manipulating Text

You can manipulate text by breaking or joining lines of text, by centering lines of text, and by changing the case of the text from upper to lowercase and vice versa.

- **Breaking Lines**
  If lines in your character session become too long, you can split them into separate lines.

  1. Place the cursor at the position where you want the break to occur.

  2. Press Ctrl-B. The session manager breaks the line, places the rest of the line on a new line, and renumbers the lines.

- **Joining Lines**
  You can join (or "glue") two consecutive lines together into a single line.

  1. Place the cursor anywhere on the first line.

  2. Press Ctrl-G. The session manager joins the lines.

- **Centering Lines**

  1. Place the cursor anywhere on the line that you want to center.

  2. Press Ctrl-6. The session manager centers the line.

- **Changing Case**

  1. Place the cursor on the character that you want to change.

  2. Press Ctrl-5. The session manager converts the case of the character and advances the cursor one position to the right.

Modifying Arrays

You can use several keystrokes to change the shape of the array you are editing. You can add a row of blank lines, add or delete rows and columns of zeros, or undo changes you make.

- **Character Session Keystrokes**
  - ❑ Press Alt-F3 or Ctrl-Ins to add a row of blanks above the cursor. If the cursor is in the bottom row, press Enter to add a row of blanks.
  - ❑ Press Alt-F4 or Ctrl-Del to delete the row that contains the cursor.
  - ❑ Press Ctrl-U to undo the most recent row deletion. You can restore up to 20 rows.
- **Numeric Session Keystrokes**
  - ❑ Press Alt-F3 or Ctrl-Ins to add a row of 0s above the cursor. If the cursor is in the bottom row, press Ctrl-Enter to add a row of 0s.
  - ❑ Press Alt-Ins to add a column of 0s to the left of the cursor. If the cursor is in the rightmost column, press Alt-Enter to add a column of 0s to the right of the array.
  - ❑ Press Alt-Del to delete the column that contains the cursor.
  - ❑ Press Ctrl-U to undo the most recent row deletions. You can restore up to 20 rows. The number of columns in deleted rows must match the current number of columns.

Changing Margins and Width

When you edit a character vector or native file, you can change the margins and edit width of the session. You cannot reformat the lines in an APL session, or when you edit a function or matrix. You can change the cell width in numeric editing sessions. You can make these changes to tagged blocks or to the entire object you are editing.

- **Character Sessions**

  1. Press Ctrl-Y to set the left margin. A message on the status line prompts you for the setting. Type the margin column number and press Enter.
  2. Press Ctrl-Z to set the right margin. A message on the status line prompts you for the setting. Type the margin column number and press Enter.
  3. Press Ctrl-W to reformat the lines within the specified margins. The session manager lengthens or shortens lines at word breaks.
  4. If the text is not tagged, a message on the status line asks you to confirm that you want to reformat the whole session. Press Y for Yes or N for No. If you have a block tagged, only that text is reformatted.

- **Numeric Sessions**

  1. Press Ctrl-W. A message on the status line prompts you for the new column width.
  2. Type the new column width and press Enter. To use the smallest possible width, type 0.

In numeric editing sessions, all columns use a single column width, based on the width of the widest number. If you add a number wider than the current column width, the column width automatically expands to accommodate it. You cannot make the column width narrower than the narrowest number.

Entering Characters You Cannot Type

Sometimes you may want to enter a character from the keyboard that you cannot type, such as a line-drawing character. To do this, you need to know the atomic vector (⎕*AV*) index of the character you want. If you do not know the index, refer to the Atomic Vector appendix in the *Quick Reference Guide*, the Atomic Vector and Keyboard Events appendix in the *Reference Manual*, or the on-line Help file..

When you know the number, follow these steps.

1. Press Shift-Esc.
2. Type the index (in origin 0).
3. Press A. The session manager enters the character in that atomic vector position as input.

You can also enter events, such as the cursor-up movement (event 393). Appendix A in the *Quick Reference Guide* and the Atomic Vector and Keyboard Events appendix in the *Reference Manual* list these events.

Editing Pendent or Suspended Functions

The full-screen editor allows you to edit the definition of a function that is suspended or pendent in the state indicator stack. However, any changes that you make to such a function do not apply to the instances on the stack unless the function is the most recent suspension. In this case, the edit changes apply only to this most recent suspension. If you restart other pendent or suspended calls to the same function, they use the original definition.

If the line number associated with a label changes in the topmost suspended function, the system updates the label to the new line number. This update does not apply to any pendent or suspended calls to the same function farther down in the state indicator stack. If you add or delete labels in any suspended function, including the topmost one, the additions or deletions do not take effect until the next time you call the function. Changing the function header has a similar effect — the new argument names or local variable names have no effect until the next time you call the function.

If you insert or delete lines in the function, the value of the line number in ⎕*SI* and ⎕*LC* does not change. If you delete lines so that the function no longer has a certain line number, ⎕*SI* and ⎕*LC* show the suspended line number as zero.

Other APL systems give the error *SI DAMAGE* if you modify the functions in the state indicator stack in a way that does not allow you to restart them. APL★PLUS II/386 cannot produce this error because changes apply only to the topmost suspended function.

Note: This behavior is subject to change in future versions.

3
Using Files

APL★PLUS II/386, like other APL★PLUS systems, gives APL the ability to store and access data in disk files outside of the APL workspace. As a result, you can use the sharing capabilities of APL★PLUS II/386 component files in conjuction with DOS files.

The following general capabilities are available with APL★PLUS II/386 disk files:

- the ability to enter, retain, and access data in disk files, where the data can remain after the APL session ends
- the ability to store more data than can fit at one time in the APL workspace and to retrieve it in manageable portions
- the ability to perform file operations under program control
- the ability to use DOS files from APL, providing an interface between APL and non-APL programs.

Basic File Operations

APL★PLUS II/386 includes a built-in file system for the APL environment. With it, you can store data either in APL component files or in DOS files on floppy or hard disks. You can store, retrieve, and manipulate data with APL system functions.

Key Concepts

To understand the material in this chapter, you should have a clear understanding of several key concepts.

- **Access**
  A user has access to an APL file when APL★PLUS II/386 permits that person to use it in some way. Typically, you have access to every file that you own, and can give access to other users by setting the **access matrix** appropriately.

- **APL Component Files**
  An APL file is structured as a set of components, each of which contains one APL array. An APL file component is much like a variable, except that it has a number instead of a name. Any value that can be stored in a variable can also be stored in a file component. The file component contains complete information about the APL array, including its shape and datatype.

- **Directories**
  DOS organizes files using a hierarchy of directories. A directory is a special kind of native file that contains a list of file names. The names in the list can be files or other directories. Any file in the system can be located by specifying its directory. You can create and access files in other directories by explicitly referring to the directory in a command.

- **Files**
  A file is a place for storing and organizing data in permanent disk storage. APL★PLUS II/386 uses two distinct kinds of files: APL component files, which hold APL arrays, and native files, which are standard DOS files.

- **Libraries**
  Traditional APL systems organize files and workspaces into libraries. Libraries are much like directories except:

  - ❑ they can be identified by numbers instead of names
  - ❑ they are not hierarchical.

  APL★PLUS II/386 provides a mechanism for simulating traditional libraries by associating a library number with a DOS directory. This behavior allows applications that were developed on other APL★PLUS systems to run without modification.

- **Native Files**
  These are DOS files. APL★PLUS II/386 treats native files as simple sequences of characters or numbers, so you are responsible for any structuring and housekeeping. You can use native files to share data with non-APL applications.

- **Ownership**
  Owning a file or workspace means that you originally created it or most recently renamed it.

- **Users**
  To APL★PLUS II/386, a user is one individual. In a network environment, every user can have an account number, which APL★PLUS II/386 uses to identify individual users for file-sharing purposes. You can discover your user account number by entering `1↑⎕AI`.

Comparing APL Workspaces and Files

There are two ways to make permanent copies of APL data: in a saved workspace or in an APL file. APL files and saved workspaces are alike in several ways.

- Both reside on a disk.
- Both can be created, used, and erased when they are no longer needed.
- Both have names and optional library numbers.
- Both can contain many objects, each of which can be any APL value. Stored data can be a scalar, vector, matrix, or higher-dimensional array, nested or simple, holding characters, numbers, or both. The only limit on the size of a component or variable is that it must fit in the available space in the active workspace and on the disk.
- Both are owned by a particular user, generally the person who created them.

There are some important differences between a file and a saved workspace.

- A workspace can contain executable programs as well as data; a file can contain only data.
- More than one file can be active at the same time.
- A file can hold amounts of data that are larger than can fit in one workspace (which is limited by the amount of memory installed in your computer). The size of a file can range from zero to many megabytes.
- APL★PLUS II/386 refers to a variable in a workspace by its name. The system refers to a component in an APL file by a component number that gives the component's position within the file. Component numbers are consecutive positive

integers ranging from the number of the first component to the number of the last component.

File System Functions

File operations bring the data in components into the active workspace for processing and save values generated in the active workspace as components of a file. File operations are performed through a collection of APL system functions. As system functions, file operations share many of the properties of APL primitive functions:

- they are always available for use in the workspace
- they can be incorporated in user-defined functions
- many return explicit results that can be used in subsequent operations.

The names of the functions used with APL files all start with quad-F (⎕*F*), and each name indicates the kind of operation being performed. Examples of such functions are ⎕*FREAD* (for reading a component from a file) and ⎕*FAPPEND* (for appending to a file a value from the active workspace). The names of the functions for use with native files all start with quad-N (⎕*N*), and each name indicates the kind of operation being performed.

Improper use of file operations can lead to file errors. Such errors are indicated by error reports. Errors generated by file functions are just like errors generated by APL primitive functions in terms of their effect on the APL statement in which they occur. Each file operation is described in detail in the section "File Sharing — Concepts and Functions," later in this chapter.

Identifying Files

You can identify files in two different ways.

- If you have an appropriate library defined, the library number and file name identify the file; for example, *3 PERSONS*.
- A complete path name, including disk drive designation and file extension, can always be used to identify a file; for example, *C:\APL\FILES\PERSONS*. The disk drive designation and directory path is optional. If it is omitted, the current directory is assumed.

When you use a file identification as an argument to a file function, represent that identification as a character vector enclosed in single quotes, or as any other APL expression whose value is a character vector. Represent a directory name as a path. If you use a library number instead, use a positive integer for the library number. You can omit the directory name or library number if it is the same as the default directory.

An APL file name can consist of from one to eight uppercase letters (*A*-*Z*) and numeric characters (0-9), and must begin with a letter; for example:

```
      'SAMPLE'
```

If you have defined a library, separate the library number from the file name with at least one space; for example:

```
      '11 TOOLS'
```

If you do not supply a library number, the system assumes the default directory on default disk drive.

Regardless of whether you define libraries, you can use the following form to identify files:

```
      'C:\JOE\USR\DATA87'
```

This form shows that the file named *DATA87* is in the subdirectory named *USR*, which in turn is in the directory named *JOE* on disk drive *C*.

If a defined library matches a file name or path, the system displays the library number; otherwise, the system displays the full path.

For maximum portability of programs to other APL★PLUS systems, the use of defined libraries and a library form of file names is recommended.

APL file identifications are patterned after workspace identifications but are logically distinct. A workspace and a file can have the same name. They are distinguished at the DOS operating system level by the file extension with which they are stored (*.SF* for files, *.WS* for workspaces), and they are distinguished within APL by the operations with which they are used.

Native file identifications use the same names both within APL and outside the APL environment. See your DOS operating system manual for the full set of conventions used by your system.

To use a file, you must pair it with an integer **file tie number**. All operations on the file refer to the file by its tie number rather than by its name. The particular value of the tie number can be any number in the acceptable range that is different from any other file tie numbers already in use. The pairing of a file and a file tie number is called a file tie; a file is said to be tied if such an association has been made.

Descriptions and examples of file operations in this chapter use the two files described in the following section. If you create those two files, you will be able to execute the examples used in the chapter.

Creating and Building Files

In the following example, the APL file named *PERSONS* has four components, each of which is a vector of characters. The APL file *SALES* also has four components, each of which is a vector of numbers.

	PERSONS	*SALES*
1	'*SMITH*'	5 6 3 1
2	'*JONES*'	2 6 1
3	'*KELLEY*'	4 6 2 9 1
4	'*BECKER*'	20 6 4

The file function ⎕*FCREATE* establishes a new APL file; the function ⎕*FAPPEND* places new components in the file. The following program builds the two files *PERSONS* and *SALES*.

```
     ∇ SAMPLE1;NAME;PMAT;T;VALUES
[1]    ⍝ CREATE THE FILES 'PERSONS' AND 'SALES'
[2]    'PERSONS' ⎕FCREATE 5
[3]    'SALES' ⎕FCREATE 20
[4]    PMAT← 4 6 ⍴'SMITH JONES KELLEYBECKER' ⋄ I←1
[5]    ⍝ PROMPT FOR SALES FOR EACH PERSON
[6]    ⍝ THEN STORE THE VALUES IN THE FILES
[7]    ⍝
[8]    LOOP:→(I>1↑⍴PMAT)/END ⋄ NAME←PMAT[I;]
[9]    'ENTER VALUES FOR ',NAME ⋄ VALUES←⎕
[10]   CN←VALUES ⎕FAPPEND 20
[11]   'VALUES STORED IN COMPONENT NUMBER ',⍕CN
[12]   ⍝APPEND TO FILE 'PERSONS'
[13]   T←(NAME~' ') ⎕FAPPEND 5
[14]   I←I+1 ⋄ →LOOP
[15]   ⍝
[16]   END: ⎕FUNTIE 20 5
     ∇
```

The ⎕*FCREATE* function creates a new file and prepares it for further operations. When ⎕*FCREATE* creates the file, it associates the file name with a tie number. You use the tie number to perform subsequent file operations. The syntax (where *fileid* stands for file identification and *tieno* stands for tie number) is:

'*fileid*' ⎕*FCREATE tieno*

The file name must be different from that of any existing file in that library, and the tie number must not be in use. When *SAMPLE*1 is executed, line [2] creates the file *PERSONS* and ties it to 5. At this point, the file *PERSONS* exists in the library, but it is empty since it contains no components. Components are added to the file *PERSONS* by the function ⎕*FAPPEND* on line [13].

Similarly, the ⎕*NCREATE* function creates a native file. Native files follow somewhat different naming conventions and are used with negative tie numbers. See ⎕*NCREATE* in the System Functions, Variables, and Constants chapter of the *Reference Manual* for more information.

The ⎕*FAPPEND* function puts a new component, containing an APL value from the active workspace, at the end of an APL file. The value in the workspace is not altered. The syntax is:

```
        compno ← value ⎕FAPPEND tieno
```

The left argument is the value for the new component. It can be the name of a variable; for example:

```
        NAMES ⎕FAPPEND 5
```

The left argument can also be the result of any APL calculation. For example:

```
        (5+2×4 6ρι24) ⎕FAPPEND 5
```

The right argument is the tie number of the file to contain the new data. The ⎕*FAPPEND* function returns the new component number as its result. Examples of appending to files *SALES* and *PERSONS* appear in lines [10] and [13] of *SAMPLE*1.

Similarly, the ⎕*NAPPEND* function appends the contents of an array, byte for byte, at the end of a native file.

Files created the way *PERSONS* and *SALES* were created have no maximum size. You can specify a size limit for the file by specifying the number of bytes after the file identification in the left argument of ⎕*FCREATE*. You will receive the error *FILE FULL* if you attempt to put more than that number of bytes of data into the file. The use of file size limits allows you to budget your disk storage and prevent "runaway" programs from filling the entire disk.

If line [2] of *SAMPLE*1 had been

```
        'PERSONS 10000' ⎕FCREATE 5
```

the size limit would be 10,000 bytes. The default value 0 indicates no size limit.

Tying and Untying Files

You can activate an existing file by tying it, as shown in the next sample function. This function prints each of the components of the file named *PERSONS* along with the sum of the numbers in the corresponding components of *SALES*. The two file functions introduced in this example are ⎕*FTIE* and ⎕*FREAD*.

```
     ∇ SAMPLE2;I;X
[1]    ⍝ TOTAL SALES FOR EACH PERSON
[2]     'PERSONS' ⎕FTIE 1
[3]     'SALES' ⎕FTIE 2
[4]     'NAME        SUM'
[5]     '----        ---' ◇ I←1
[6]    ⍝
[7]    LOOP:→(I>4)/END ◇ X←⎕FREAD 2,I
[8]     (10↑⎕FREAD 1,I),⍕+/X
[9]     I←I+1 ◇ →LOOP
[10]   ⍝
[11]   END: 'COMPLETE.'
     ∇
```

The ⎕*FTIE* function associates a file identification with the tie number; this process is called tying a file. In *SAMPLE*2, *PERSONS* is tied to tie number 1 and *SALES* is tied to tie number 2. The syntax of ⎕*FTIE* is:

```
        'fileid' ⎕FTIE tieno
```

The function ⎕*FTIE* has the same effect as ⎕*FCREATE*, except that the file named in the left argument must already exist. *SAMPLE*1 ties the file *PERSONS* to tie number 5, while *SAMPLE*2 ties it to 1. File tie numbers used with ⎕*FTIE* can be any integer from 1 through 2147483647, and the number for a particular file can be different on different occasions.

Similarly, tie numbers for use with ⎕*NTIE* can be any integer from ¯1 through ¯2147483648. You must use negative numbers as tie numbers for native files.

The function *SAMPLE2* tied the files *PERSONS* and *SALES* but did not untie them, so those two files remain tied as files 1 and 2. A file remains tied until you untie it or end the APL session. Thereafter, you must re-establish the ties to use the same files.

The status of tied files is not changed by any system commands except)*OFF*. Therefore, loading into or clearing the active workspace leaves active any ties set previously during the session. This behavior means that you can load different workspaces without having to re-tie the files.

The ⎕*FUNTIE* function breaks the association of an APL file and tie number. It has the following syntax:

```
        ⎕FUNTIE  tienos
```

The argument is a vector of 0 or more file tie numbers, so ⎕*FUNTIE* can untie several files at once (as in line [16] of *SAMPLE1*). The ⎕*FUNTIE* function has no effect on the values stored in a file.

Similarly, the ⎕*NUNTIE* function breaks the pairing of a native file and its tie number.

Reading Data from a File

The ⎕*FREAD* function reads the value of a file component into the active workspace. The value can be placed in a variable (as in line [7] of *SAMPLE2*), used in an expression (as in line [8]), or displayed directly at the terminal. The syntax of ⎕*FREAD* (where *compno* stands for component number) is:

```
        result ← ⎕FREAD  tieno compno
```

The argument is a two-element vector. The first element is the tie number and the second element is the component number. The ⎕*FREAD* function returns the value of the specified file component as its explicit result.

Reading from a native file with the ⎕*NREAD* function is more complicated, since the file is not logically divided into components and does not internally track the datatype of what has been stored there. See ⎕*NREAD* in the System Functions, Variables, and Constants chapter of the *Reference Manual* for more information.

Erasing Files

When you no longer need an APL file, you can erase it with the ⎕*FERASE* function. The syntax is:

fileid ⎕*FERASE* *tieno*

The ⎕*FERASE* function deletes the file from the library. All of its components are destroyed and the space they occupied is made available for use by other files. Since the file no longer exists, the file tie is also broken.

You must tie a file before you can erase it, and you must specify both the file name and the tie number exactly as you specified it when you tied the file.

You can use the ⎕*NERASE* function to erase native files.

Copying Values

The following function shows how to change the arrangement of filed information. The components of *PERSONS* and *SALES* are to be merged into a new file *RECORDS*. Each odd-numbered component will come from *SALES*. The two original files are erased after the merge is complete. Recall that *PERSONS* and *SALES* are still tied following execution of *SAMPLE2*.

```
     ∇ SAMPLE3;T
[1]    ⍝MERGE FILES TIED TO 1 AND 2 INTO 'RECORDS'
[2]     'RECORDS' ⎕FCREATE 3 ⋄ I←1
[3]    ■
[4]    LOOP:→(I>4)/END
[5]     T←(⎕FREAD 1,I) ⎕FAPPEND 3 ⍝ FROM 'PERSONS'
[6]     T←(⎕FREAD 2,I) ⎕FAPPEND 3 ⍝ FROM 'SALES'
[7]     I←I+1 ⋄ →LOOP
[8]    ⍝
[9]     END:⎕FUNTIE 3
[10]    'PERSONS' ⎕FERASE 1 ⋄ 'SALES' ⎕FERASE 2
     ∇
```

Note that when the files have been merged, the new file *RECORDS* (tied to 3) is untied. It is not necessary to untie the files tied to 1 and 2 because erasing the files unties them automatically.

Listing Files in a File Library

The ⎕*FLIB* function lists the names of the files in a particular library or directory. The syntax is:

```
result ← ⎕FLIB lib
```

where *lib* stands for library number or directory name. The *result* is a character matrix. Each row holds the identification of a file in the designated library. If, for example, the files in program *SAMPLE*1 had been created by a user in library *C*:\JOE\USR, then the expressions

```
⎕LIBD '3 C:\JOE\USR'
⎕FLIB 3
```

produce

```
3 PERSONS
3 SALES
```

All of the APL files in a library or directory appear in the result of ⎕*FLIB*, even those that the user account number requesting the display is not authorized to use.

The expression

```
        ⎕FLIB ⍳0
```

lists the APL files in the current default library (working directory). The system command *)FLIB* yields the same information as *⎕FLIB*.

Similarly, *⎕LIB* and *)LIB* return a list of all files (native, APL, and even workspaces) in a format consistent with native file names.

File Management Facilities

This section covers the remaining file functions used with both shared and nonshared files. The examples in this section assume that you can create and tie files. Unlike the examples in the previous section, you should not run these examples directly from your keyboard.

Inquiring about File Ties

Two system functions, ⎕*FNAMES* and ⎕*FNUMS*, report the names and numbers of the current APL file ties. The ⎕*FNAMES* function returns a character matrix in which each row is the file identification of a currently tied APL file. The ⎕*FNUMS* function returns a vector of current tie numbers, in the same order as the names returned by ⎕*FNAMES*.

For a tabular display of this information, use:

```
      ⎕FNAMES,⎕FNUMS
SAMPLE          1
PERSONS         2
```

To untie all tied APL files, execute the statement:

```
      ⎕FUNTIE ⎕FNUMS
```

The vector returned by ⎕*FNUMS* is exactly what ⎕*FUNTIE* needs to untie all files.

Similarly, the ⎕*NNAMES* and ⎕*NNUMS* functions report on the status of native file ties. Given the negative tie numbers for native files, the corresponding formula for a tabular display is:

```
      ⎕NNAMES,⎕NNUMS
```

Replacing a File Component

You can replace a file component with a new APL value using the function *⎕FREPLACE*. The syntax is:

```
        value  ⎕FREPLACE  tieno  compno
```

For example, to replace the second component of the file tied to 1 with the character vector *'REPLACED'*, use the following statement:

```
        'REPLACED' ⎕FREPLACE 1 2
```

The new value can be any APL value that fits in the active workspace. If the new value is larger than the previous value, the system uses additional file storage automatically.

A common use of *⎕FREPLACE* is to update a component by catenating a new value to it. In this example, the value of the variable *DEB* is catenated to the existing value of component 3 of the file tied to 7:

```
        ((⎕FREAD 7 3),DEB) ⎕FREPLACE 7 3
```

Note that the new value is larger than the value it is replacing. A replacement component need not occupy the same amount of space as the old value.

The corresponding native file operation, *⎕NREPLACE*, does not use components. Rather than a component number, you supply the position in the file where the replacement begins. That value replaces exactly the amount of the file needed to store it, regardless of what was there before. You must evaluate positioning and space requirements. If you try to store more data than the remaining space in the file can accommodate, *⎕NREPLACE* lengthens the file to make room for the rest of the data. For additional details, see *⎕NREPLACE* in the System Functions, Variables, and Constants chapter of the *Reference Manual*.

Dropping Components

The ⎕*FDROP* function removes components from either end of an APL file. The syntax is:

```
        ⎕FDROP tieno n
```

The argument to ⎕*FDROP* is a two-element vector. The first element is the tie number of the file. The second element is an integer that specifies both the number of components to be dropped, and from which end of the file to drop them.

- If *n* is positive, ⎕*FDROP* removes the components from the front of the file.
- If *n* is negative, ⎕*FDROP* removes the components from the end of the file.
- If *n* is 0, ⎕*FDROP* removes no components.

The component numbers of the remaining components are not changed. For example, if the file tied to 99 has 10 components numbered 1 through 10, after running

```
        ⎕FDROP 99 4
```

the file has six components remaining, numbered 5, 6, 7, 8, 9, and 10. If you run

```
        ⎕FDROP 99 ¯2
```

a second time, the file has components 5, 6, 7, and 8.

You cannot drop components from the interior of a file. You can use other techniques to signify that an interior component holds no information and should be bypassed in later processing. One way is to replace the component with an empty vector (' ').

There is no native file operation that corresponds to ⎕*FDROP*, since native files are not organized into components.

Determining the Size of a File

The ⎕*FSIZE* function displays size information about a file. Its syntax is:

```
        result ← ⎕FSIZE tieno
```

The result is a four-element vector. The first two elements are the component limits of the file: the number of the first component, and a number that is one higher than the number of the last component. The component limits of a newly created file are 1 1.

Use the following expression to display a count of the number of components in the file:

```
        |-/2↑⎕FSIZE tieno
```

The third and fourth elements of the result are the amount of file storage currently occupied by the file and the file storage limit, in bytes. If the fourth element is 0, the file has no imposed size limit other than available space on the disk.

The fifth element of the result is the number of bytes of allocated but unused file storage occupied by the file. You can reclaim this space with the ⎕*FDUP* function. (See "Compacting a File," later in this chapter.)

The corresponding native file operation, ⎕*NSIZE*, returns a single number representing the number of bytes in use. Since native files have neither components nor automatic checking for maximum size, there can be no meaningful equivalents for the other numbers in ⎕*FSIZE*.

Suppose the result of ⎕*FSIZE* for the file tied to 10 is:

```
        ⎕FSIZE 10
      1 126 259584 265000
```

The result of ⎕*FSIZE* indicates that the components in the file are numbered 1 through 125, and that the size limit for the file is 265,000 bytes, of which a total of 259,584 bytes are now occupied.

The following examples show how to implement two common file organizations — "first-in, first-out" (FIFO) and "last-in, first-out" (LIFO) — using *⎕FDROP* and *⎕FSIZE*. The examples use a file tied to 50.

FIFO Organization

Data collection:	*data* *⎕FAPPEND* 50 *SIZE* ← *⎕FSIZE* 50
Processing:	*INFO* ← *⎕FREAD* 50,*SIZE*[1] (process info) *⎕FDROP* 50 1

LIFO Organization

Data collection:	*data* *⎕FAPPEND* 50 *SIZE* ← *⎕FSIZE* 50
Processing:	*INFO* ← *⎕FREAD* 50,*SIZE*[2]-1 (process info) *⎕FDROP* 50 ¯1

With each of these schemes, the data collection and processing phases can be executed when it is convenient to do so. You can collect data whenever information is available and process it when the file becomes large and unwieldy, or at some fixed interval; for example, once a day or once a week.

After a component is processed, the storage it occupies is generally released by executing *⎕FDROP*. In some cases, you may need to use *⎕FDUP* to reclaim all dropped storage; see the System Functions, Variables, and Constants chapter of the *Reference Manual*. Consequently, data can be collected and processed almost indefinitely and never require more than a relatively small amount of storage space.

The next example shows a function that uses *⎕FDROP* with a file whose components are released in an arbitrary sequence. *SAMPLE4* produces an invoice from information in a file named *CDATA*. After producing the invoice, *SAMPLE4* drops the appropriate component of *CDATA* (if it is the first component) or replaces it with an empty component (if it is not the first component). Each time you run *SAMPLE4*, the function checks to see if the last component of data is an empty vector. If it is, *SAMPLE4* drops that component. This technique tends to minimize the file size.

```
     ∇ SAMPLE4;LIM;N
[1]    'CDATA' ⎕FTIE 5
[2]    'ENTER RECORD NUMBER' ⋄ N←⎕
[3]    ⍝ TEST COMPONENT NUMBER:
[4]    LIM←2↑⎕FSIZE 5 ⋄ →((N<LIM[1])∨N≥LIM[2])/ERR
[5]    ⍝ READ RECORD AND PRODUCE INVOICE:
[6]    RECORD←⎕FREAD 5,N ⋄ →(0=×/⍴RECORD)/ERR
[7]    PRINTINVOICE RECORD
[8]    '' ⎕FREPLACE 5,N
[9]    ⍝ DROP ANY EMPTY COMPONENTS
[10]   ⍝ FROM FRONT OF FILE
[11]   LIM←2↑⎕FSIZE 5
[12]   ⍝TEST FOR EMPTY FILE
[13]   LOOP:→(LIM[1]=LIM[2])/END
[14]   ⍝ TEST FOR EMPTY COMPONENT
[15]   →(0≠×/⍴⎕FREAD 5,LIM[2]-1)/END
[16]   ⍝ DROP EMPTY COMPONENT
[17]   ⎕FDROP 5 ¯1 ⋄ LIM[2]←LIM[2]-1 ⋄ →LOOP
[18]   ⍝
[19]   ERR:'RECORD NUMBER NOT IN FILE'
[19]   END:⎕FUNTIE 5
     ∇
```

In line [4] of *SAMPLE4*, a branch is taken to *ERR* if *N* is not within the range of the lowest and highest component numbers. Otherwise, line [6] reads the component, branching to *ERR* if the component is empty. Line [7] calls the function that produces an invoice based on the record. Lines [11] through [17] drop empty components, if any, from the end of file. Line [13] checks for an empty file.

The preceding discussion shows one possible file organization technique. There are more static forms of database file organization that subdivide the data and keep directories. You may also want to use the convention of reserving the first component for a file description.

Renaming a File

The ⎕*FRENAME* function changes an APL file name. The syntax is:

```
'fileid' ⎕FRENAME tieno
```

You can use ⎕*FRENAME* to change the name of a file in a library:

```
'REPORTS' ⎕FTIE 100
'OLDRPTS' ⎕FRENAME 100
⎕FUNTIE 100
```

The ⎕*FRENAME* function does not create a second copy of a file. After executing the previous example, *REPORTS* disappears from the library.

In addition to changing the name, ⎕*FRENAME* also sets the ownership of the file to match the user account number of the person who performed the operation.

The system function ⎕*NRENAME* is used for renaming native files and for moving a native file into a different directory.

Changing a File's Size Limit

The ⎕*FRESIZE* function changes the size limit on an APL file. The syntax is:

```
newsize ⎕FRESIZE tieno
```

where *newsize* is the new file size limit.

You can use ⎕*FRESIZE* in three ways: to impose a size limit on file, to increase a size limit, or to decrease a size limit.

Sometimes a file needs to hold more data than its current size limit allows. When a file does not have enough room for the value to be stored in it, a *FILE FULL* error occurs:

```
      'OLDRPTS' ⎕FTIE 12
      REPORT ⎕FAPPEND 12
FILE FULL
      REPORT ⎕FAPPEND 12
      ^       ^
```

The fourth element of the result of ⎕FSIZE shows that the size limit for the file *OLDRPTS* is 100,000:

```
      ⎕FSIZE 12
1 34 99512 100000
```

The ⎕FRESIZE function can increase the file size limit so that there will be enough room to append the new data:

```
      200000 ⎕FRESIZE 12
      ⎕FSIZE 12
1 34 99512 200000
      REPORT ⎕FAPPEND 12
34
```

You can also decrease the size limit of a file, provided you do not specify less storage than is already being used by the data in the file. In addition, you can remove the size limit restriction completely:

```
      0 ⎕FRESIZE 12
      ⎕FSIZE 12
1 34 99512 0
```

There is no equivalent capability for native files.

Copying a File

The ⎕*FDUP* function makes a duplicate copy of the contents of an APL file in another APL file. The copy need not be in the same library or directory as the file being copied, and the name need not be the same.

```
      '5440 FIGHT' ⎕FTIE 48
      '1844 OREGON' ⎕FDUP 48
      ⎕FUNTIE 48
```

The ⎕*FDUP* function does not change the ownership of the original file, but the user account number that used ⎕*FDUP* is the owner of the copy.

You need not untie the new copy because it was never tied. If you tie the new copy and compare its size to the size of the original file, you may find that the new file is smaller. The ⎕*FDUP* function eliminates any wasted space while copying (see "Compacting a File," below).

There is no comparable system function for duplicating native files, but you can use the DOS COPY command, either from DOS or with)*CMD*, or under program control with ⎕*CMD*. This method can also be used to copy any file, including APL files and saved workspaces, without compacting them. In fact, because ⎕*FDUP* always compacts a file, you should not use it for routine file copies. Use the DOS COPY command instead.

To move a file from one library to another, use ⎕*FRENAME* for APL files and ⎕*NRENAME* for native files.

Compacting a File

You can also use ⎕*FDUP* to compact the contents of an APL file.

```
'5440 FIGHT' ⎕FTIE 1844
'5440 FIGHT' ⎕FDUP 1844
⎕FUNTIE 1844
```

When you use ⎕*FDUP* to compact a file, you must tie the file exclusively; that is, use ⎕*FTIE*. The user account number that used ⎕*FDUP* to compact the file becomes the owner of the file.

Since storage space is limited, you may need to reclaim file space containing data abandoned by the use of ⎕*FDROP* or ⎕*FREPLACE*. APL★PLUS II/386 does not automatically release all the space occupied by abandoned data within APL files. You must reclaim storage space as needed by using ⎕*FDUP*.

There is no system function for compacting native files, since the system does no tracking of their contents.

Getting Information about Files and File Components

The ⎕*FHIST* function provides information about the history of an APL component file. Its syntax is:

```
⎕FHIST  tieno
```

The function returns a three-row matrix with the following information:

- Row 1 contains the user number of the file owner and the timestamp of the file's creation in both packed form and ⎕*TS* form.
- Row 2 contains the user number and timestamp associated with the most recent change to the file.

- Row 3 contains the user number and timestamp associated with the most recent setting of the file access matrix.

There is no corresponding function for native files.

Four pieces of information accompany the value in each component of a file. They are:

- the workspace storage needed to hold the component's value
- the user account number of the person who last replaced or appended the component
- the time that the component was last replaced or appended, in a packed-timestamp form
- the component timestamp in a seven-element unpacked form (year, month, date, hour, minute, second, millisecond).

The ⎕*FRDCI* function reads component information. Its syntax is:

result ← ⎕*FRDCI tieno compno*

The result of ⎕*FRDCI* is a ten-element numeric vector holding the component information. The packed timestamp in the third element of the result is given in microseconds since 00:00, 1 January 1900.

⎕*FRDCI* is particularly useful in data collection or audit trail applications, where several users may have access to add data to a file. As each user adds data, the component is automatically tagged with the time and the user account number. Later, when a user with ⎕*FREAD* and ⎕*FRDCI* authorization processes the file, the source of each component is clear.

Sharing Files among Users

There are two basic ways to share files among multiple users: exclusively and concurrently. When users share a file exclusively, one user has access to the file at a time. No other user can work with the file until the current user is done. When users share a file concurrently, all the users can work with the file at the same time.

This section describes the mechanisms you can use to control concurrent file sharing and limit other users' access to your files.

Concurrent Sharing of Files

Two file system functions allow concurrent file sharing: *⎕FSTIE* and *⎕FHOLD*.

The *⎕FSTIE* function ties a file, exactly as *⎕FTIE* does. However, this "shared tie" also permits others to share-tie the same file. The syntax is:

```
'fileid' ⎕FSTIE  tieno
```

You can use *⎕FSTIE* to share-tie a file if the file allows any form of access, as long as no other user has the file exclusively tied using *⎕FTIE*.

When several users use a file concurrently, their file operations proceed asynchronously. One user's processing may be interleaved with another's. For example, suppose that users P and Q have a file tied to 77 and are trying to add 1 to the value of component 5. They use the following statement:

```
(1+⎕FREAD 77 5) ⎕FREPLACE 77 5
```

When both users finish, component 5 should be increased by 2. However, if the parts of the statement are executed in the following sequence, the value of component 5 is increased by only 1, since P's program destroyed the value stored by Q.

```
User P                        User Q

                              ⎕FREAD 77 5
⎕FREAD 77 5
                              (add 1)
                              ⎕FREPLACE 77 5
(add 1)
⎕FREPLACE 77 5
```

An interlock can prevent P's program from executing any part of the APL statement while Q's program is executing it (and vice versa).

The *⎕FHOLD* function provides this interlock. Its syntax is:

⎕FHOLD tienos

The *⎕FHOLD* function places an interlock on each file whose tie number is included in the right argument. The concept of this interlock is subtle; in effect, using *⎕FHOLD* means "wait until no one else has these files held, then hold them for me."

As executed by P, *⎕FHOLD* has three effects.

- Any interlocks in effect from a previous *⎕FHOLD* executed by P are released (even if they held the same file).
- P is placed in a queue behind every other user who has already executed *⎕FHOLD* for any of the files specified by P. P's program is delayed until no ther user holds any of these files.
- Interlocks are set simultaneously on all of the designated files, and P's program then resumes execution.

File holds do not prevent others from using the file while it is held. A file hold only delays the execution of *⎕FHOLD* in other users' programs. In other words, no two users can have the same file held at the same time. File holds do not block other file operations such as *⎕FREAD* or *⎕FREPLACE*.

⎕FHOLD does provide a means for two or more users to cooperate and avoid conflict in file use. For example, if the program executed by P and Q in the previous example is changed to:

```
      ⎕FHOLD 77
      (1+⎕FREAD 77 5) ⎕FREPLACE 77 5
      ⎕FHOLD ⍳0
```

The first user to execute *⎕FHOLD* 77 can proceed without delay, while the other user's program is delayed until the first user's program executes *⎕FHOLD* ⍳0. The sequence looks like this:

User P	**User Q**
	⎕FHOLD 77
⎕FHOLD 77	
(delay)	(proceed)
	⎕FREAD 77 5
	(add 1)
	⎕FREPLACE 77 5
	⎕FHOLD ⍳0
(proceed)	(proceed)
⎕FREAD 77 5	
(add 1)	
⎕FREPLACE 77 5	
⎕FHOLD ⍳0	
(proceed)	

When Q executes *⎕FHOLD* 77, the hold on the file prevents P from establishing a hold. In P's program, *⎕FHOLD* simply waits to hold the file, which happens when Q executes *⎕FHOLD* ⍳0. P can then hold the file and execution proceeds. Cooperating users can therefore avoid conflict.

However, if Q does not use *⎕FHOLD* properly, then P cannot prevent conflict. Suppose P is using *⎕FHOLD*,

```
      ⎕FHOLD 77
      (1+⎕FREAD 77 5) ⎕FREPLACE 77 5
      ⎕FHOLD ⍳0
```

but Q is **not** using *⎕FHOLD*:

```
      (1+⎕FREAD 77 5) ⎕FREPLACE 77 5
```

The interaction is as if P were not using ⎕*FHOLD* at all:

User P	**User Q**
⎕*FHOLD* 77	
(proceed)	⎕*FREAD* 77 5
⎕*FREAD* 77 5	(add 1)
(add 1)	⎕*FREPLACE* 77 5
⎕*FREPLACE* 77 5	
⎕*FHOLD* ⍳0	

Because users must cooperate for ⎕*FHOLD* to work, you may want to use a specific function to access a file. You can use a file passnumber (discussed in the next section) to enforce the use of a given protocol.

All interlocks are released when the user who set them executes another ⎕*FHOLD*, signs off, or enters immediate execution mode. Untying or retying a file releases any interlock set on it. ⎕*FHOLD* ⍳0 releases all interlocks.

The immediate execution case is particularly important to remember. If you type the following three statements as three different immediate execution inputs, the ⎕*FHOLD* has no effect at all.

```
      ⎕FHOLD 1
      PROCESS 1
      ⎕FHOLD ⍳0
```

However, a compound statement works correctly, since there is no immediate execution input between ⎕*FHOLD* and process:

```
      ⎕FHOLD 1 ⋄ PROCESS 1 ⋄ ⎕FHOLD ⍳0
```

The following examples illustrate the use of file holds.

- Several users are concurrently appending to a file but make no other use of the file. File holds are not needed since ⎕*FAPPEND* tracks components added to the file by number. Although you cannot predict the order of the various values, any request for a file operation always waits until a previous operation on the same file is complete. The result of

⎕*FAPPEND* lets each user know which component contains his or her value.

- Several users are reading and replacing components of a file. No two users ever reference the same component. For example, P's program refers only to component 1, Q's program only to component 2, and so on. No hold is necessary, since no conflict can occur on the concurrent use of a single component.

- An application involves the use of three files in which like-numbered components contain related data. One program updates the files while the other programs read the files concurrently. To ensure that no program reading from the files encounters a mixture of old and new data, the updating program has this appearance:

```
        ⎕FHOLD 21 22 23
(update code)
        ⎕FHOLD ⍳0
```

  The file-reading programs that are operating concurrently have this appearance:

```
        ⎕FHOLD 21 22 23
        A ← ⎕FREAD 21,N
        B ← ⎕FREAD 22,N
        C ← ⎕FREAD 23,N
        ⎕FHOLD ⍳0
```

  This example shows a program that needs file holds even though the program itself is not modifying the contents of any files.

As these examples show, the need for file holds depends upon the interrelation of program and file structure. The design of any application involving concurrent use of files requires careful analysis for possible "race conditions" between programs. You can resolve such conflicts with appropriate use of ⎕*FHOLD*.

The File Access Matrix

Associated with every APL file is an access matrix that records the users who are authorized to use the file and the APL file operations that each can perform. The access matrix affects only the actions taken by an APL user with APL file system functions. It offers no protection against use by non-APL programs or from APL programs that have tied the file as a native file.

Every access matrix is an integer-valued matrix with three columns and any number of rows.

- Column 1 is the user number you want to give access. A zero gives access to all users.
- Column 2 indicates the operations a user is can perform.
- Column 3 is a passnumber to the file.

For example, suppose you want to set up a file that allows all users only ⎕*FREAD* access. You want to be able to test the application that accesses the file and obtain the same permission as another user. You would set up an access matrix like this:

0	1	0	⎕*FREAD*-only access to all but owner.
owner	1	0	⎕*FREAD*-only access to owner.
owner	¯1	1234	Full access with passnumber to owner to perform other file operations.

Table 3-1 shows typical entries in an access matrix. The details of columns 2 and 3 are of interest only when exercising detailed control over access; they are discussed later in this section.

Table 3-1. Typical Access Matrix Entries

Account Number	Encoding of Permitted Operations	Passnumber	Description
12345	¯1	0	Access granted to user 12345.
23456	9	5	Access granted to user 23456.
0	1	0	Access granted to all other users.

Access matrices follow these rules:

- Every access matrix is an integer matrix with three columns and any number of rows. Each row in the access matrix represents the authorization granted for a single user or class of users. The first column of the matrix contains a user number; the second column contains an encoding of the corresponding authorized operations; and the third column contains a passnumber.
- The user numbers in Column 1 can be those of any users. A 0 value in Column 1 refers to all users other than the owner or those explicitly specified elsewhere in Column 1.

  The owner has full implicit access only if that user number does not explicitly appear in Column 1 of the access matrix. If the owner does appear explicitly in the access matrix, then access is granted according to the access matrix.

- The value in Column 2 explicitly indicates which operations the user is authorized to perform. Each operation subject to access control is associated with a value, called the access code, that is a power of 2. The sum of these access codes is the nominal value in Column 2.

  Formally, the value in Column 2 is the integer representation of a Boolean mask that has a bit for each controllable operation. If

  ```
          MASK ←→ (32⍴2)⊤VALUE
  ```

  then *MASK*[32-*N*] (origin 1) is the bit regulating the operation whose access code is 2⋆*N*. If the bit is 1, the user is authorized for the operation. Many bit positions are not associated with any operations; the value of these bits is immaterial. Thus, ¯1 grants authorization for every operation, since (32⍴2)⊤¯1 ←→ 32⍴1.

  This last property is often used to grant all but certain kinds of access to a file. The technique is to subtract from ¯1 the access codes for the operations to be denied. For example, ¯5 grants all but ⎕*FERASE* access (¯1-4); ¯7 grants all but ⎕*FERASE* access; ⎕*FTIE* (¯1-(4+2)).

- The passnumber in Column 3 can be any positive or negative integer value. Together with the user number in Column 1, the passnumber determines which row of the access matrix is applied in verifying authorization for each operation. A single user or class of users can have multiple rows in the access matrix, with different privileges granted with differing passnumbers.

Access matrix settings do not restrict functions like ⎕*FNAMES* and ⎕*FNUMS*. These functions never produce a *FILE ACCESS ERROR* because they do not work on a particular file. The ⎕*FSIZE*, ⎕*FHIST*, or ⎕*FSTIE* functions have no limitations either; these are permitted if any other file operation is authorized.

Reading the Access Matrix

The ⎕*FRDAC* function reads the access matrix into the active workspace. The syntax is:

```
        result ← ⎕FRDAC  tieno
```

The result is the access matrix for the file tied to the number in the right argument. The second element in a row of the access matrix is the sum of the access code values for the particular file operations that are allowed for the user account number in the first position of the row. Table 3-2 shows the access code values.

Use the sum of all possible access codes to authorize all possible file operations. You can also use ¯1 to grant full access, as well as access to future file operations. A value of 9 indicates authorization to use the functions ⎕*FREAD* and ⎕*FAPPEND* on the file. Any nonzero value permits use of ⎕*FSIZE*, ⎕*FHIST*, and ⎕*FSTIE*. **Note:** not all combinations of values make sense.

Table 3-2. Access Code Values

Code	Operation
1	⎕*FREAD*
2	⎕*FTIE*
4	⎕*FERASE*
8	⎕*FAPPEND*
16	⎕*FREPLACE*
32	⎕*FDROP*
64	Not used
128	⎕*FRENAME*
256	Not used
512	⎕*FRDCI*
1024	⎕*FRESIZE*
2048	⎕*FHOLD*
4096	⎕*FRDAC*
8192	⎕*FSTAC*
16384	⎕*FDUP*
32768	Not used
65536	Not used, represents ⎕*FSTATUS* on other systems

Setting the Access Matrix

The ⎕*FSTAC* function sets the value of a file's access matrix. The syntax is:

```
        matrix ⎕FSTAC tieno
```

The left argument is an integer-valued, three-column matrix. A three-element integer vector is reshaped to become a one-row matrix. It replaces the previous value of the access matrix. Initially, when a file is created, its access matrix has no rows: its shape is 0 3. The file owner has complete access since the user account number does not appear in Column 1 and no other user has any access for the same reason.

Changing the Access Matrix When a File is Tied

In APL★PLUS II/386, access to an APL file is determined only when the file is tied. Consequently, changing the access matrix of an APL file while it is tied has no immediate effect on access. The new value of the access matrix is used when another user tries to tie the file.

Similarly, if you change the access matrix of a file in a way that limits or otherwise changes access to the file, the change does not affect the access until the next time you re-tie the file. Therefore, changing file access to prevent the file owner from erasing the file gives no protection until the owner unties or re-ties the file.

Overriding an Access Matrix

Under the rules of access control, you can be locked out of one of your own files. Since the ability to set the access matrix is one of the operations governed by the access matrix, you may be unable to correct the problem with *⎕FSTAC* alone. The *FILEHELPER* function in the *UTILITY* workspace allows you to access a file you own but cannot access.

Using Passnumbers

The third element in a row of the access matrix, the passnumber, is usually 0. Nonzero passnumbers are used only to exercise detailed control over file access. If the third element contains a passnumber other than 0, the user whose account number is on that row must provide a matching passnumber to perform file operations.

Omitting a passnumber from an argument is equivalent to providing an explicit passnumber of 0. A mismatching passnumber causes a *FILE ACCESS ERROR*.

The passnumber used to tie a file remains associated with the file tie. After you tie a file, you must supply the same passnumber in all subsequent operations on the file that you supplied when you first tied the file. Using another passnumber always produces a *FILE ACCESS ERROR*, even if some row of the access matrix happens to grant the appropriate permission with that passnumber. To use the permission granted by a different row of the access matrix, you must re-tie the file using that row's passnumber.

Passnumbers are intended for use within locked APL functions that are used in place of standard file functions. Suppose each component of the personnel file *2345 PERS* is a vector holding an employee's telephone extension and room number. The user with account number 9876 is allowed to retrieve the telephone and room number, but not to change it.

You choose a passnumber, 10349, and define these locked functions:

```
      PTIE
[1]      '2345 PERS' ⎕FSTIE 25 10349

      R←PREAD N
[1]      R←2↑⎕FREAD 25,N,10349
```

Next, you set the access matrix for 2345 *PERS* to authorize read-only access by user 9876. The row of the access matrix is:

```
9876 1 10349
```

Finally, you give the locked functions to user 9876, but you do not reveal the passnumber. Now, user 9876 can tie and read from the file, but only by specifying the passnumber with each ⎕*FSTIE* or ⎕*FREAD*. Since the user does not know the passnumber, access to the file 2345 *PERS* is possible only through the functions *PTIE* and *PREAD*.

Using this technique, you can impose complex restrictions on file authorization; in fact, you can impose any sort of restriction that you can state as an APL function. Since you can impose different passnumbers on different user account numbers, you can provide multiple levels of access authorization to confidential data. The previous example showed access restricted to reading a file. Examples of other forms of control follow.

- Access only to even-numbered components

  ```
  [1] ⎕ERROR (0≠2|N)/'EVEN NUMBERED CNS ONLY' ◊
  R←⎕FREAD25,N,32049
  ```

- Automatic logging of information requests

  ```
  [1] A←N ⎕FAPPEND 99 2888 ◊ R←⎕FREAD 25,N,32049 ◊
  ⍝ LOG TO FILE 99
  ```

  Later use of ⎕*FREAD* and ⎕*FRDCI* on the file tied to 99 gives the value of *n*, the timestamp, and the requester's account number.

- **Access only after validating data**

```
[1] →(1≠ρρV)/ERR ◊ →((20≠ρV)∨0∨.>V)/ERR ◊ R←V
⎕FAPPEND 45 14149
```

- **Access only after verifying user identity by questions and answers (to protect a momentarily unattended computer from passersby).**

- **Access only to summary data (for instance, salary averages but no individual salaries).**

Comparing APL and Native Files

The correspondence between APL file operations and native file operations is not always exact. In particular, note that:

- Native files are always used with directory names. Library numbers do not apply to native files.
- You use negative tie numbers with native files; you use positive tie numbers with APL files.
- The ⎕*NREAD* function needs a more complex right argument since the file is not organized into components.
- The ⎕*NAPPEND* and ⎕*NREPLACE* functions place the exact contents of the array into the file, retaining no information about the array such as shape or datatype. Nested or heterogeneous arrays are not allowed.

You should compare the APL component files and the native files available through APL★PLUS II/386 in some detail. APL files provide more automatic housekeeping and control and greater convenience when making changes. Native files permit an easy interface with non-APL systems such as word processors, since these are the files that non-APL programs use.

Files Are a Sequence of Stored Data Items

Both APL files and native files can be viewed as a simple sequence of stored data items. They differ as follows.

- The APL file is a sequence of APL arrays. Each array, independent of the others, can be of any datatype, of any rank, shape, or size. One can be a table of decimal numbers, while the next can be a four-page memo. Regardless of the nature or size of the array, you can refer to it by a single component number and retrieve it by that number. When you retrieve a component, the system automatically recognizes and handles the array's internal organization (the number of bytes per element and the interpretation of the arrangement of the bits) and external organization.

- The native files created by *⎕NCREATE* are sequential DOS files. They contain a sequence of bytes (characters or numbers). You must determine the organization and representation of the data. This represents maximum flexibility at the cost of maximum programming effort. The program performing the retrieval must deal with where to start reading the material, how far to read before reaching the other end, and whether to reshape the data and how.

Space Reservation and Checking

An APL file can be given a size limit. Each time you attempt to add or replace material in the file, the system automatically checks whether the operation would cause the file to exceed the limit. If the limit would be exceeded, the operation is not performed, a *FILE FULL* error is signaled, and the programmer (or the program) must deal with that condition before continuing. This permits budgeting of disk space and recognition of "runaway" programs.

There is no such size checking in writing to a native file.

Ownership and Access Control

APL★PLUS II/386 tracks the account number that created or last renamed the file (its owner). Through the use of the access matrix, the file owner and other authorized users can extend or limit the types of file operations that a given account number can perform. You can use nonzero passnumbers in the access matrix to limit operations; for example, you can restrict use to locked APL functions or special kinds of ties.

Access to native DOS files is controlled by a less specific mechanism. DOS open modes and byte range locks can be used to control access for native files. Also, the DOS ATTRIB command can be used to control the type of access permitted to a file.

Replacing Data

You can replace one component in an APL file with any other APL array using ⎕FREPLACE. You need not match the physical size in bytes, the datatype, or the shape of the array being replaced. In a file with 20 components, you could replace the integer in component 7 by a table of numbers without stopping to ask how much room on the file the integer occupied:

```
      ASTRO ⎕FREPLACE 5 7
```

Consequently, it is easy to replace an object with an enlarged (updated) version of itself; for example, a customer list with one or more new customers added.

In a native file, you can only replace data byte-for-byte. You (or the program) must determine the size of the data to be put into the file, and how that compares with the amount already there. You must also determine where the data begins, what information will be destroyed, whether space will be wasted, and additional space requirements.

An updated list or otherwise enlarged object cannot simply replace an earlier version in the file. To preserve the same relative position entails copying the rest of the file with the new data into a new file. To avoid copying all of the data, you must set up a directory system to track the data. This directory can include currently unused space abandoned as data grows. If you do not track unused space, you cannot reclaim it.

If an APL file is cramped for space, reclaiming wasted space is simple. The ⎕FDUP function uses the internal component-tracking data to compact the file to only its useful data. Using ⎕FDUP requires that enough space is available on the disk to hold all of the compacted data.

Since native files do not predictably include such tracking information, you must provide the means for compressing the space.

Different Naming Conventions

The names of APL files are restricted to combinations of alphabetic and numeric characters, while the names of native files can contain other characters permitted by DOS naming conventions (see your DOS operating system manual) and are not restricted to names beginning with a letter.

4

Formatting Data

This chapter describes the formatting capabilities of the system function ⎕*FMT* and gives an overview of the workspace *FORMAT*. The chapter is organized as follows.

- "Designing a Report" discusses the decisions you need to make before you try to format your data.
- "What is ⎕*FMT*?" defines the system function ⎕*FMT* and explains its syntax and arguments.
- "How to Construct a Format String" explains the different parts of a format string.
- "Editing Format Phrases" explains how to use the format phrases that edit data.
- "Positioning and Text Format Phrases" explains how to use the format phrases that position fields and insert text.
- "Parameters" explains how to use formatting parameters to give detailed instructions to ⎕*FMT*.
- "Grouping Symbols" explains how to simplify formatting.
- "Modifiers" explains how to use formatting modifiers to add decorations and special effects to formatted data.
- "Stars or Underscores in Result" describes the formatting errors that cause stars or underscores in the result.
- "Useful Applications" lists common w
- "Using the *FORMAT* Workspace" is an *FORMAT* workspace.

Designing a Report

Before you generate a report, you should design it. You must consider several factors before you format the data for your report:

- the order of the columns
- the numeric representation of the columns
- the width of the columns
- the width of the entire report.

You also must decide how you want to clarify the meaning of the data. For example, you may want to include dollar signs, commas, negative signs, and so on. In some cases, a pattern may be the best way to represent data; for example, you can place parentheses around the area code of a telephone number. Figure 4-1 shows the structure of a typical report.

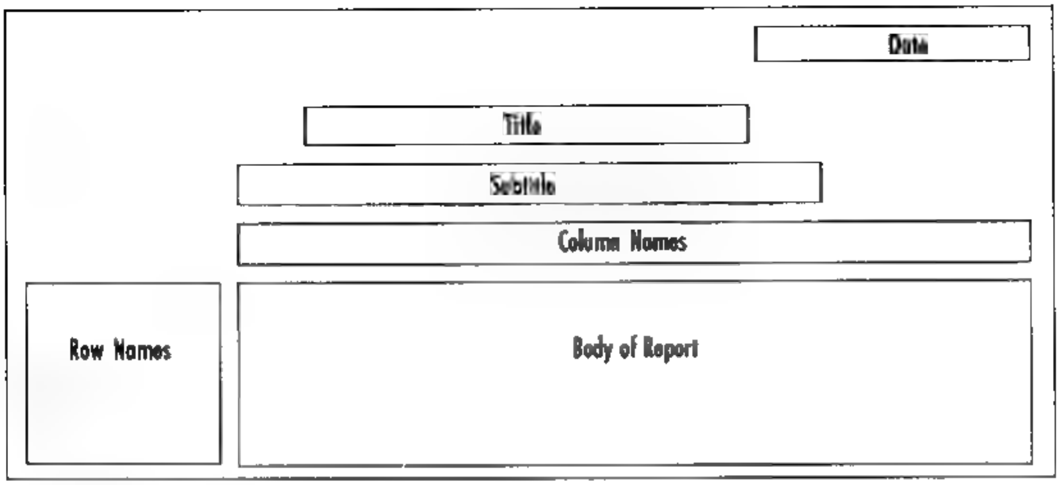


Figure 4-1. Design of a Typical Report

Once you decide on the basic layout of your report, use the functions described in the section "Using the *FORMAT* Workspace" to add and position titles and labels. The following report is an example of what a typical report might look like using the structure shown in Figure 4-1.

```
                                                              1/5/87
                            HARRIS GARAGE
                 EASTERN DIVISION INVENTORY REPORT
PART NAME   NUMBER    QUAN   PRICE     VALUE         REORDER    <6MO
BATTERY     879-01    492    $92.85   $45,682.20     12/13/87
CARBURETOR  657-04    769    $73.23   $56,313.87     5/06/87       R
FUEL TANK   876-03    371    $71.80   $26,637.80     6/24/87
WHEEL       234-06    287    $41.75   $11,982.25     4/12/87       R
BATTERY     876-07    381    $96.45   $36,747.45     9/03/87
TIRE        876-02    98     $60.90    $5,968.20     4/25/87       R
AXLE        265-07    205    $55.85   $11,449.25     11/13/87
TIRE        361-08    387    $66.95   $25,909.65     9/26/87
BRAKE       876-06    201    $32.00    $6,432.00     3/01/87       R
CARBURETOR  876-04    879   $157.80  $138,706.20     6/25/87
TIRE        234-01    298    $68.90   $20,532.20     2/11/87       R
EXHAUST     876-05    357   $354.00  $126,615.00     6/25/87
IGNITION    876-09    652    $22.50   $14,670.00     3/12/87       R
SPARK PLUG  273-03    391     $2.85    $1,114.35     8/05/87
RADIATOR    872-05    738    $63.80   $47,084.40     2/28/87       R
WATER PUMP  251-09    276    $53.78   $14,843.28     9/08/87
ALTERNATOR  729-07    493    $96.70   $47,673.10     7/14/87
RADIATOR    316-02    387    $69.30   $26,819.10     10/26/87
COIL        582-08    492    $25.50   $12,546.00     1/21/87       R
```

The report above was produced with the program *INV*. The program uses the system function ⎕*FMT* and several functions from the *FORMAT* workspace (see next page).

The program defines a format string containing formatting instructions, assigns the data variables, uses the title functions and column and row name functions to set the titles and label the rows and columns, and finally calls ⎕*FMT* to format the data into the report.

```
     ∇ INV;COLNAME;DATE;FS;NUM;PRICE;QUANT;RN;VALUE
[1]   FS←'13A1,T16,G<Z99-99>,I6,T29,P<$>F8.2,T38'
[2]   FS←FS,',CP<$>F12.2,T51,G<Z9/99/99>'
[3]   FS←FS,',N< R>Q<     >I6,X6,4< >'
[4]   NUM← 87901 65704 87603 23406 87607 87602 26507
[5]   NUM←NUM, 36108 87606 87604 23401 87605 87609
[6]   NUM←NUM, 27305 87205 25109 72907 31602 58208
[7]   RN←'/BATTERY/CARBURETOR/FUEL TANK/WHEEL/BATTERY'
[8]   RN←RN,'/TIRE/AXLE/TIRE/BRAKE/CARBURETOR/TIRE'
[9]   RN←RN,'/EXHAUST/IGNITION WIRE/SPARK PLUG/RADIATOR'
[10]  RN←RN,'/WATER PUMP/ALTERNATOR/RADIATOR/COIL'
[11]  QUANT←492 769 371 287 381 98 205 387 201 879 298
[12]  QUANT←QUANT,367 652 391 738 276 493 387 492
[13]  PRICE←92.85 73.23 71.8 41.75 96.45 60.9 55.85
[14]  PRICE←PRICE,66.95 32 157.8 68.9 345 22.5 2.85
[15]  PRICE←PRICE,63.8 53.78 96.7 69.3 25.5
[16]  DATE←121387 50687 62487 41287 90387 42587 111387
[17]  DATE←DATE,92687 30187 62587 21187 625487 31287
[18]  DATE←DATE,80597 22887 90887 71487 102687 12187
[19]  VALUE←QUANT×PRICE
[20]  COLNAME←'/PART NAME/NUMBER/QUAN/PRICE/VALUE'
[21]  COLNAME←COLNAME,'/REORDER/<6MO.'
[22]  COLNAME←FS COLNAMES COLNAME
[23]  RN←(⍳0) ROWNAMES RN
[24]  FS RJUST '1/5/87'
[25]  FS CENTER 'HARRIS GARAGE'
[26]  FS CENTER 'EASTERN DIVISION INVENTORY REPORT'
[27]  ''
[28]  COLNAME
[29]  ''
[30]  FS ⎕FMT(FN;NUM;QUANT;PRICE;VALUE;DATE;×DATE-60587)
     ∇
```

What Is ⎕*FMT*?

The system function ⎕*FMT* is a tool for detailed tabular formatting, which allows you to:

- format several data arguments at a time
- vary the printing order of the columns of data from the normal left-to-right order
- insert message text
- display tables of numbers in designated rows and columns
- decorate monetary values with dollar signs or other identifiers
- express the results of floating-point calculations as standard decimals or integers
- place flags to mark negative or zero values in results.

⎕*FMT* is a dyadic system function. Its syntax is either

```
        result ← 'fstring' ⎕FMT data
```

or, in strand form,

```
        result ← 'fstring' ⎕FMT data1 data2 ... datan
```

The left argument, *fstring*, is a character vector that specifies how you want the data to look. It contains specific formatting instructions that control the editing and display of the right argument, the data you want to format. A data item in the right argument can be the name of a variable or an APL expression that produces a result.

The result of ⎕*FMT* is always a character matrix. It can be stored and used within a larger expression like any other APL expression. For example, the result of the expression

```
        'I4' ⎕FMT 987
```

is a one-row, four-column character matrix containing the value 987. The number of rows in the matrix result is determined by the maximum number of matrix rows or vector elements in the data. The number of columns is determined by both the format string and the right argument.

The value of ⎕*PW* does not control the size of the formatted result, and the value of ⎕*PP* does not affect precision.

Right Argument — the Data List

The right argument to ⎕*FMT* is a data list containing APL variables, constants, or expressions that return numeric or character scalars, vectors, or matrices. A single data expression in the right argument needs no parentheses; for example:

```
        'fstring' ⎕FMT data
```

To format two or more data expressions, the right argument can be a nested vector, like those formed by strand notation:

```
        'fstring' ⎕FMT data1 data2 ...datan
```

If you are writing code that is to run on APL★PLUS systems that do not have nested arrays, you can separate the expressions with semicolons and enclose the entire set in parentheses:

```
        'fstring' ⎕FMT (data1;data2;...;datan)
```

You can replace any item in the data list with an expression that produces the desired value as a result (though you may need parentheses). The following examples show some permissible right arguments:

```
SCALAR ← 15
VEC ← 3.5 4E3 0.007 1
CHAR ← 'MONDAY'
MAT ← 3 4 ρι12

'fs' ⎕FMT SCALAR
'fs' ⎕FMT 2 2 ρVEC
'fs' ⎕FMT (SCALAR;VEC;CHAR;MAT)
'fs' ⎕FMT SCALAR VEC CHAR MAT
'fs' ⎕FMT (MAT;+/MAT)
'fs' ⎕FMT MAT (+/MAT)
```

APL★PLUS II/386 formats data expressions of different shapes as follows:

- a matrix has each column formatted separately
- a vector is treated as a one-column matrix
- a scalar is treated as a one-row, one-column matrix.

If you want to display a vector horizontally, reshape it as a one-row matrix. Each column of data (which can be one scalar, one vector, or one column of a matrix) is formatted individually as specified by the left argument.

Left Argument — the Format String

The left argument to ⎕*FMT* is a character vector containing one or more format phrases. There are two classes of format phrases: editing format phrases and positioning format phrases.

- **Editing Format Phrases**
  These phrases edit data in the right argument; for example, the *I* format phrase displays numeric data in integer format.

- **Positioning and Text Format Phrases**
  These phrases change the appearance of the result without editing any data; for example, the *T* format phrase specifies tab stops for column placement.

You can also use special **parameters** with many of the format phrases to specify the field width and precision of the image, and you can use **modifiers** to add special effects.

Separate individual format phrases with commas and enclose the entire string in single quotes (the phrase must be character-valued). You can use the format string directly as a character vector, or you can store it in a variable and use it later; for example,

```
      'I4' ⎕FMT V
```

or

```
      F ← 'I4'
      F ⎕FMT V
```

Text Delimiters

Place text in the left argument between pairs of delimiters. Any of the pairs of delimiters shown in Table 4-1 are valid. The closing delimiter must correspond to the opening one.

Table 4-1. Valid Text Delimiters

Delimiters	Examples
< >	<19__>
⊂ ⊃	⊂CR⊃
¨ ¨	¨ = ¨
⎕ ⎕	⎕-⎕
⍞ ⍞	⍞YES⍞
/ /	/NO/

You can use any of the delimiters as text characters within a format phrase; but one of them cannot be the closing delimiter. For example, to use the characters

```
5>3?
```

as text in a format phrase, you could enclose them in different delimiters:

```
⊂5>3?⊃
```

How the Format String is Processed

The format string is scanned phrase-by-phrase from left to right. Editing format phrases are matched with columns of data, and positioning and text phrases are processed as they are encountered without reference to the data.

You need not provide the same number of format phrases as columns of data in the right argument. The number of format phrases does not have to divide the number of columns of data evenly. If the format string contains an insufficient number of phrases to edit all of the data in the right argument, ⎕*FMT* cycles through the format string repeatedly until it edits all of the data. If the format string contains more phrases than are necessary to edit all of the data, the trailing phrases are ignored.

Spaces in the format string have no effect, except between text delimiters.

How to Construct a Format String

A format string comprises several parts.

- **Format Phrases**
  Format phrases edit and position text and data.
- **Parameters**
  Parameters, which are used with format phrases, give ⎕*FMT* detailed instructions.
- **Modifiers**
  Modifiers add special effects and decorations to edited data.

For example, in the format string

```
CI13
```

I is the format phrase that tells ⎕*FMT* to format numeric data as integers, 13 is the parameter to the *I* format phrase that specifies a field width of 13 columns, and *C* is a modifier that specifies that commas should separate every three digits in the data.

To construct a format string, follow these steps.

1. Determine a report width and the width of individual fields within the report.
2. Select appropriate format phrases for the corresponding data.
3. Code the required parameters, such as the field width and the number of digits to be displayed.
4. Include modifiers and decorations for special effects.
5. Add positioning phrases to the format string to control the location of fields.

To simplify the coding of repeated format phrases or groups of format phrases, use repetition factors within parentheses. A repetition factor is a non-negative integer that indicates the number of times that ⎕*FMT* applies a phrase or a group of phrases. The default repetition factor is 1.

You may find it helpful to store format strings as variables, especially if you will be using them with several different right arguments.

The Editing Format Phrases

The *A*, *I*, *F*, *E*, and *G* format phrases place columns of data in the right argument into corresponding fields in the result.

A — Character Editing

The *A* format phrase is the only phrase that edits only character data. The *A* phrase takes the form

Aw

where *w* is the field width. The field width specifies the number of columns in the result to be occupied by the edited value. (Remember that a vector is treated as a one-column matrix and is formatted as a single column, unless it is reshaped to a one-row matrix.)

```
      'A1' ⎕FMT 'SALT'
S
A
L
T
```

A character matrix in the data can contain several columns. In that case, you must specify a separate format phrase for each column. For example,

```
      CMATRIX ← 1 4 ⍴'SALT'
      'A1,A1,A1,A1' ⎕FMT CMATRIX
SALT
```

Instead of typing the format phrase repeatedly, you can use a repetition factor to repeat the phrase. In the following example, a repetition factor of 4 is used.

```
      '4A1' ⎕FMT CMATRIX
SALT
```

When the same format phrase is used for the entire right argument, as in the above example, you can omit the repetition factor.

```
      'A1' ⎕FMT CMATRIX
SALT
```

If the field width is greater than 1, leading positions of the field are filled with blanks.

```
      WORDSQ ← 3 3 ρ'BOAURNSET'
      'A1, A2, A3' ⎕FMT WORDSQ
B O  A
U R  N
S E  T

      'A2, A2, A2' ⎕FMT WORDSQ
 B O A
 U R N
 S E T
```

If you try to use an *A* format phrase to edit numeric data in the right argument, the result field is filled with stars (*).

```
      'A6' ⎕FMT 133
******
```

Stars also result if you try to use an edit phrase other than *A* to edit character data.

```
      'I4' ⎕FMT 'ABC'
****
****
****
```

I — Integer Editing

The *I* format phrase edits numeric data in integer format. The *I* phrase takes the form

```
      Iw
```

where *w* is the field width. An *I* format phrase displays numeric data as integers. Be sure to include space in the field width for negative signs in the result that correspond to negative values in the data.

```
      TABLE ← ¯3 2 ¯1 ∘.* ¯2 ¯1 1 7
      TABLE
0.1111111111 ¯0.3333333333 ¯3 ¯2187
0.25           0.5          2   128
1             ¯1           ¯1    ¯1

      '3I4, I6' ⎕FMT TABLE
0   0  ¯3 ¯2187
0   1   2   128
1  ¯1  ¯1    ¯1

      'I8' ⎕FMT (3×4), 6E2 0.4 476.85912
      12
     600
       0
     477
```

F — Fixed Point Editing

The *F* format phrase edits numeric data in fixed-point or decimal format. The *F* phrase takes the form

Fw.d

where *w* is the field width and *d* is the number of decimal positions. For example, the format phrase F9.2 creates a field of nine positions, in which every value is formatted to two decimal places. Allow two positions in the field width for a possible negative sign and a decimal point.

```
      'F9.2' ⎕FMT TABLE
      0.11     ¯0.33     ¯3.00 ¯2187.00
      0.25      0.50      2.00   128.00
      1.00     ¯1.00     ¯1.00    ¯1.00
```

For numbers between ¯1 and 1, ⎕FMT places a 0 to the left of the decimal point in the result.

E — Exponential Editing

The *E* format phrase edits numeric data in exponential (scientific notation) format. The *E* phrase takes the form

Ew.s

where *w* is the field width and *s* is the exact number of significant figures in the nonexponent part of the result. For example, the format phrase *E*15.7 produces a field width of 15 with 7 significant digits. The number of significant digits specified must be between 1 and 17, inclusive. The width must be at least seven more than the number of significant digits to provide space for a negative sign, a decimal point, and a five-position exponent of the form *E*¯*nnn*. Additional width leaves visible space between columns.

```
      'E10.2' ⎕FMT TABLE
  1.1E¯1   ¯3.3E¯1   ¯3.0E0    ¯2.2E3
  2.5E¯1    5.0E¯1    2.0E0     1.3E2
  1.0E0    ¯1.0E0    ¯1.0E0    ¯1.0E0

      'E12.4' ⎕FMT 2*50 ¯50
   1.126E15
   8.882E¯16
```

G — Pattern Editing

The *G* format phrase edits numeric data according to a pattern or picture format. With this phrase, you can arbitrarily mix text characters and data in a formatted column by creating a pictorial template for the data. Special characters within the pattern indicate where to display digits in the data; other characters are displayed as they appear in the pattern. The *G* phrase takes the form

G<pattern>

where *pattern* consists of the special characters 9 and Z, called digit selectors, along with other text characters you insert in the field. You can use a pair of text delimiters to enclose the pattern (see the section "Right Argument — the Data List"). For example, to format a date with a G pattern, enter:

```
      'G<99/99/99>' ⎕FMT 11887
01/18/87
```

The number of characters, including blanks, between the pattern delimiters determines the field width. The field is formatted with the exact number of characters and digits specified in the pattern.

Data in the right argument is rounded to the nearest integer. Each digit of the integer is transferred to the field to replace a digit selector (9 or Z) in the pattern. A 9 in the pattern transfers the corresponding digit from the integer into the result. A Z suppresses leading and trailing 0s, transferring the corresponding digit only if the digit is not 0 or if it is between two transferred digits, as described below.

Use the Z digit selector to omit leading or trailing 0s in the edited fields. If a Z corresponds to a 0 in the integer, the 0 is transferred only if digits on both sides are transferred; otherwise, the corresponding position in the field is unchanged. Since a 9 always transfers a digit to the result, a Z between two 9s acts as a 9 digit selector. For example, `G<Z9Z.99>` is equivalent to `G<Z99.99>`. Table 4-2 compares the effects of using 9s and Zs as digit selectors.

Table 4-2. Using 9s and Zs as Digit Selectors

Format Phrase	Data	Result
`G<Z99.99>`	`2460 1200 0`	`24.60 12.00 00.00`
`G<Z99.9Z>`	`2460 1200 0`	`24.6  12.0  00.0`
`G<ZZ9.9Z>`	`2460 1200 0`	`24.6  12.0   0.0`
`G<ZZ9.ZZ>`	`2460 1200 0`	`24.6  12      0`
`G<ZZZ.ZZ>`	`2460 1200 0`	`24.6  12`

You must insert text characters or Z digit selectors in the pattern to display negative signs and decorations (decorations are explained in the section "Modifiers" in this chapter). Leading and trailing text characters always transfer directly to the result. Text between digit selectors transfers only if digits to the right and to the left of the pattern text are transferred. Compare the appearance of 0s and commas in the next two examples.

```
      NUM ← 298738472 389487.987 387 0.35
      Z ← 'G<$ ZZZ,ZZZ,ZZ9 AND NO CENTS>'
      N ← 'G<$ 999,999,999 AND NO CENTS>'

      Z ⎕FMT NUM
$ 298,738,472 AND NO CENTS
$     389,488 AND NO CENTS
$         387 AND NO CENTS
$           0 AND NO CENTS

      N ⎕FMT NUM
$ 298,738,472 AND NO CENTS
$ 000,389,488 AND NO CENTS
$ 000,000,387 AND NO CENTS
$ 000,000,000 AND NO CENTS
```

You can use G formatting to produce visually effective reports when numbers are displayed in traditional patterns.

```
      EMPLO←4 7⍴'ABEL   GALOIS GAUSS  ZORN   '
      SSN←298374562 298750385 384716453 273069857
      TEL←4086729873 7187364782 8063948726 3138472637
      SALES ← 567 309 4958 312
      SAL ← 49800 50000 59500 41200

      FS←'7A1,G<999-99-9999 >,G< (999) 999-9999>'
      FS ← FS,',I6,G<  ZZ,ZZ9>'
      FS ⎕FMT (EMPLO;SSN;TEL;SALES;SAL)
ABEL   298-37-4562  (408) 672-9873   567  49,800
GALOIS 298-75-0385  (718) 736-4782   309  50,000
GAUSS  384-71-6453  (806) 394-8726  4958  59,500
ZORN   273-06-9857  (313) 847-2637   312  41,200
```

Although data values edited by a *G* format phrase are rounded to integers, fractional values in the data can be displayed by multiplying the data by the appropriate power of 10. In the following example, the data are multiplied by 100.

```
      MONEY ← 1 4 ⍴14.6 52 17 2.44
      'G<  $99.99>' ⎕FMT 100×MONEY
$14.60  $52.00  $17.00  $02.44
```

A better way to achieve the same effect is to prefix the *G* format phrase with the *K* scaling modifier (see the section "Modifiers" in this chapter). In the following example, the data are scaled by `10*2`.

```
      'K2G<  $99.99>' ⎕FMT MONEY
$14.60  $52.00  $17.00  $02.44
```

The Positioning and Text Format Phrases

The positioning format phrases allow you to position fields without having to count individual positions. The positioning phrase specifies the column where the result of the next format phrase should begin.

- The *T* phrase specifies the starting column relative to the left margin.
- The *X* phrase specifies the starting column relative to the current position.

The *<text>* format phrase inserts text directly into the result field. These three phrases do not edit data; they simply position it in the result.

T — Absolute Tabbing

The *T* phrase specifies absolute tabbing (jumping to a particular column). It takes the form

Tp

or

T

where *p* is the column position (counting from the left margin) at which to format the next edited value. For example, *T*15 moves directly to position 15 regardless of the previous position. The first available position on a line is *T*1.

If you use *T* without specifying the position, the next field is formatted to the right of the rightmost column of the result formatted so far.

```
      '3A1,T10,I2' ⎕FMT(1 3 ρ'TAB';10)
TAB      10
```

Using a *T* phrase before individual format phrases can make the format string easier to modify. The position of a field is clear and is independent of previous formatting instructions.

X — Relative Tabbing

The *X* phrase specifies relative tabbing, or positions to be skipped. It takes the form

```
      Xp
```

where *p* is the number of positions to be skipped from the present position before formatting the next field of the result.

- If *p* is positive, *p* is the number of positions to move right.
- If *p* is negative, *p* is the number of positions to move left.
- If *p* is 0, the phrase is ignored.

To specify a negative value, use a negative sign (`X¯12`) or a minus sign (`X-12`).

```
      'I12,X¯12,2I4,X4,I4' ⎕FMT 1 4ρι4
2   3   1   4
```

<text> — Text Insertion

The *<text>* phrase inserts the text between the delimiters into a line. The phrase takes the form

```
      <text>
```

where *text* is any combination of characters and spaces. All of the text between the delimiters, including blanks, is inserted directly into the edited line. You can use any valid pair of text delimiters to enclose the text (see the section "Right Argument — The Data List" in this chapter).

```
      DPT ← 3 4 5
      REV ← 344.50 89.74 250.13
      FS←'<REVENUE FOR DEPT.>,I2,
< IS..$>,F6.2'

      FS ⎕FMT (DPT;REV)
REVENUE FOR DEPT. 3 IS..$344.50
REVENUE FOR DEPT. 4 IS..$ 89.74
REVENUE FOR DEPT. 5 IS..$250.13
```

Using Positioning and Text Phrases

You can use the *T* and *X* format phrases to position fields without having to count individual spaces. In the format string

```
      I5, X3, 25A1, T51, 4F7.2
```

the first floating point field begins in position 51, and if all four *F* phrases are used, the result is a matrix with 78 (50+4×7) columns.

Positioning phrases may reorder data in the result. The next example uses parentheses and repetition factors to simplify the left argument (see the sections "The Positioning and Text Format Phrases" and "Parameters" in this chapter).

```
      FS ← '3(A1, <-    >),T1,3(I3,X2)'
      FS ⎕FMT (3 3ρ'ABCDEFGHI';3 3ρι9)
A-1  B-2  C-3
D-4  E-5  F-6
G-7  H-8  I-9
```

You can use text phrases to override previously formatted fields. The decimal point is replaced by a colon in this example.

```
      FS ← 'F7.2, X¯3, <:>, T'
      FS ⎕FMT 1 4ρ10+0.15×¯1+ι4
10:00 10:15 10:30 10:45
```

You can use symbol substitution to achieve the same effect (see the section "Modifiers" in this chapter).

A backward-pointing relative or absolute tab may cause a previously formatted field to be overlaid by a new field. This new field need not match the width or alignment of any previously formatted field. In this case, nonblank characters in the new field replace the corresponding characters in the old field. Blank characters in the new field that occur as the result of explicit mention in *<text>* phrases, *G* phrases, or certain modifiers also replace the corresponding characters in the old field. However, blanks used as fill characters do not replace any characters in the old field.

In the next example, the background fill modifier *R* alters the normal blank background fill character (see the section "Modifiers" in this chapter). The blanks in the rightmost *R* modifier override previously formatted fields because they are text characters rather than fill characters.

```
      IOTA ← 1 4 ⍴1 2 3 4
      'I12,T1,2I4,T,I4' ⎕FMT IOTA
   2   3   1   4

      'R<•>I12,T1,2I4,T,I4' ⎕FMT IOTA
•••2•••3•••1   4

      'R<•>I12,T1,2R< >I4,T,I4' ⎕FMT IOTA
   2   3•••1   4
```

Parameters

You can use special parameters within format phrases to give ⎕*FMT* detailed instructions. Use them to specify:

- the number of times you want a phrase or a group of phrases applied
- the width of an edited column
- the number of digits to be displayed
- the position for a tab or skip.

Most parameters specify essential formatting considerations and are required with their corresponding format phrases; however, repetition factors are optional. Repetition factors improve the readability of a format string that contains repeated format phrases and often clarify the structure of the format string at a glance. They are useful when a long format phrase is needed.

w — Field Width Parameter

The field width parameter *w* determines the number of positions in the result to be occupied by the edited value. It is required with the following phrases: *A* as in *Aw*, *E* as in *Ew.s*, *F* as in *Fw.d*, and *I* as in *Iw*.

```
      'F6.1,F10.1,F8.1,F12.1' ⎕FMT 2 4
ρ58.8
  58.8      58.8    58.8          58.8
  58.8      58.8    58.8          58.8

      A←6 4ρ'OPTSPOSTPOTSSPOTSTOPTOPS'
      'A1' ⎕FMT A
OPTS
POST
POTS
SPOT
STOP
TOPS
```

If the field width is greater than the number of digits or characters, the field is padded with leading blanks.

```
      'A2' ⎕FMT A
 O P T S
 P O S T
 P O T S
 S P O T
 S T O P
 T O P S
```

Be sure to leave space in the field width for:

- a negative sign in an *E*, *F*, or *I* phrase
- a decimal point in an *F* phrase, or in an *E* phrase if more than one significant digit is specified
- an exponent of the form *E* ¯*nnn* in an *E* phrase
- any spacing you might want between the columns of a matrix.

If the field width is insufficient, the field is filled with stars (see the section "Stars or Unknown Digits in Result" in this chapter). For example:

```
      TABLE ← ¯3 2 ¯1 ∘.* ¯2 ¯1 17
      'F5.1' ⎕FMT TABLE
0.1 ¯0.3*****
0.3  0.5*****
1.0 ¯1.0 ¯1.0
```

d — Decimal Position Parameter

The decimal position parameter *d* controls the number of digits that appear to the right of the decimal point. It is required with *F* phrases (*Fw.d*).

```
      FS ← 'F2.0, F7.3, F10.6, F12.9'

      FS ⎕FMT 1 4⍴○1
3.  3.142  3.141593  3.141592654
```

s — Significant Digits Parameter

The significant digits parameter *s* controls the number of significant digits in the nonexponent part of the result. The value of this parameter must be at least 1. This parameter is required with *E* phrases (*Ew.s*).

```
      'E10.2,E9.1,E17.9'⎕FMT 1 3 ⍴¯57.98765
   ¯5.8E1   ¯6E1       ¯5.79876500E1
```

<pattern> — Pattern Text Parameter

The pattern text parameter *<pattern>* provides detailed control over the location of the individual digits of the value to be formatted. It is required with *G* phrases (*G<pattern>*). See the section "The Editing Format Phrases" in this chapter for detailed information on this parameter.

p — Position Parameter

The position parameter *p* controls the location at which to display the next field. It is required with *X* phrases (*Xp*) but is optional with *T* phrases (*Tp*). When you use it with an *X* phrase, *p* is the number of positions to be skipped from the present position. For example, *X*1 says to skip the first position and start with the second. The *p* parameter can be a positive or negative integer, or 0. A 0 causes ⎕*FMT* to ignore the phrase. You can specify a negative value with either a negative sign (*X*¯12) or a minus sign (*X*-12).

When used with a *T* phrase, *p* is the number of positions from the left margin and is always a positive integer. For example, *T*1 indicates the first position.

```
      'I12, 4(X¯4, I1)' ⎕FMT 1 4 ρι4
4   3   2   1

      FS ← 'T12,I1,T9,I1,T6,I1,T3,I1'
      FS ← ⎕FMT 1 4ρι4
4   3   2   1
```

r — Repetition Factor

The optional repetition factor *r* determines how many times to use a single format phrase or a group of format phrases enclosed in parentheses. You can use it with any format phrase. The repetition factor precedes all other elements in the format phrase. For example, a repetition factor of 2 to the left of an editing format phrase causes that phrase to edit two successive columns from the data list. For example, `F6.3, F6.3` is the same as `2F6.3`.

A nonzero repetition factor to the left of a *T* phrase has no effect. For example, `3T5` is the same as `T5` and `3T` is the same as `T`.

A nonzero repetition factor to the left of an *X* phrase repeats the specified skip. For example, `3X5` is the same as `X15` and `3X¯5` is the same as `X¯15`.

A nonzero repetition factor to the left of a *<text>* phrase repeats that phrase. For example, `2<PAGO  >` is the same as `<PAGO  PAGO  >`.

A 0 repetition factor to the left of any phrase causes `⎕FMT` to ignore that phrase. The default repetition factor is 1.

Grouping Symbols

Use parentheses in the left argument of ⎕*FMT* to simplify the construction of repeated sets of format phrases and to limit further scanning of the format string when the data has been exhausted.

Enclose the group of format phrases in parentheses and place the repetition factor to the left of the left parenthesis.

```
12A1, 4(F10.2, I4)
I15, 4(X¯4, I1)
```

Without parentheses, you would have to specify the second format string in the preceding example as:

```
I15,X¯4,I1,X¯4,I1,X¯4,I1,X¯4,I1
```

A 0 repetition factor to the left of a left parenthesis causes ⎕*FMT* to ignore the phrases in the group.

If you use parentheses and do not specify a repetition factor, 1 is assumed.

After ⎕*FMT* uses all columns of data, it continues to use any *T*, *X*, and *<text>* format phrases in the format string until:

- it finds an editing phrase (*A*, *E*, *F*, *G*, or *I*) with a nonzero repetition factor
- it finds a right parenthesis with a repetition factor that has not been fully used
- it finds a left parenthesis
- the end of the format string.

The next example demonstrates how you nest parentheses and how you use them to limit scanning of a format phrase when there is no data left to edit.

```
ADD ← 2 4 ρ1 2 3 4 5 6 7 8
ADD←(+/ADD),ADD
```

```
          'I2,< = >,4(I1,(<+>))' ⎕FMT ADD
10 = 1+2+3+4
26 = 7+6+7+8
```

Modifiers

Use modifiers to add decorations and special effects to edited data. Place them between the repetition factor and the format phrase. You can use any number of modifiers in any order.

B — Blank

The *B* modifier leaves the field blank if the edited value is 0. Use the *B* modifier with *F*, *G*, and *I* format phrases.

```
      EX ← ¯65423.43 ¯10 ¯0.4 0 100
      'BF10.1' ⎕FMT EX
   ¯65423.4
      ¯10.0
       ¯0.4
    (0 shows as blank)
      100.0

      'BG<ZZZZZ99>' ⎕FMT EX
¯65423
    ¯10
    (0.4 shows as blank)
    (0 shows as blank)
    100
```

C — Comma

The *C* modifier inserts commas between each group of three digits in the integer part of the edited value. Use the *C* modifier with *F* and *I* format phrases. Remember to provide extra positions in the field width for commas.

```
      DATA ← 987309 3870.23 74382 38E5
      'CI13' ⎕FMT DATA
     987,309
       3,870
      74,382
   3,800,000
```

K — Scaling

The *K* modifier scales (multiplies) a number before displaying it. It takes the form

```
      Ki
```

where *i* is a positive or negative integer, or 0. A negative value can be specified by a negative sign (*K*¯2) or a minus sign (*K*−2). Each number to which the *K* modifier applies is multiplied by 10⋆*i* before it is formatted.

```
      'F8.2,K1F10.2,K¯2F8.2'⎕FMT 1 3 ρ470.6
  470.60   4706.00     4.71
```

Use the *K* modifier with *E*, *F*, *G*, and *I* phrases. With an *F*, *G*, or *I* phrase, the *K* modifier controls how far the digits are shifted left or right of the decimal point in the result. With an *E* phrase, the *K* modifier adjusts the exponent in the result.

Table 4-3 shows the use of scaling with decimal, exponential, and integer editing.

Scaling is particularly useful when formatting numbers with the *G* phrase, as shown in Table 4-4.

Table 4-3. Using Scaling with *F*, *E*, and *I* Phrases

Format Phrase	Number	Result
*F*6.2	24.60	24.60
*K*1*F*6.2	24.60	246.00
K¯2*F*6.2	24.60	0.25
*E*10.4	24.60	2.460*E*1
*K*1*E*10.4	24.60	2.460*E*2
K¯2*E*10.4	24.60	2.460*E*¯1
*I*3	24.60	25
*K*1*I*3	24.60	246
K¯2*I*3	24.60	0

Table 4-4. Scaling with the *G* Phrase

Format Phrase	Number	Result
G<9.*ZZ*>	24.60	0.25
*K*1*G*<*Z*9.*ZZ*>	24.60	2.46
*K*2*G*<*ZZ*999>	24.60	2460
*K*2*G*<*ZZ*.99>	24.60	24.60
*K*2*G*<*ZZ*.9*Z*>	24.60	24.6

L — Left Justify

The *L* modifier left-justifies the edited value in the result field. Use the *L* modifier with *F* and *I* format phrases.

```
      'LI9' ⎕FMT 2*⍳8
2
4
8
16
32
64
128
256
```

When numbers with fixed decimal points or negative signs are left-justified, the alignment may be unusual.

```
      EX ← 34.5 266 0.300 ¯49.04
      'LF10.2' ⎕FMT EX
34.50
266.00
0.30
¯49.04
```

M — Negative Left Decoration

The *M* modifier replaces negative signs with text you specify. It takes the form

M<text>

where *text* replaces the negative sign to the left of the result. Use the *M* modifier with *F*, *I*, and *G* phrases. Be sure to provide space in the field width for the decoration text. *M*<¯> is the default negative left decoration.

In the next example, the *M* modifier replaces the APL negative sign (¯) by the minus sign (−).

```
      EX ← ¯65423.43 ¯10 ¯0.4 0 100
      'M<->F10.1' ⎕FMT EX
65423.4
  -10.0
   -0.4
    0.0
  100.0
```

N — Negative Right Decoration

The *N* modifier places text you specify to the immediate right of an edited negative value. It takes the form

N<text>

where *text* represents the text. Use the *N* modifier with *F*, *I*, and *G* phrases. Be sure to provide space in the field width for the decoration text.

```
      'N<  MINUS>F20.2' ⎕FMT E
   ¯65423.43  MINUS
      ¯10.00  MINUS
       ¯0.40  MINUS
               0.00
             100.00
```

O — Format Special Value as Text

The *O* modifier places the text that you specify into fields that format as 0 or that match a special value. It takes the form

```
        Ovalue<text>
```

where *text* represents the text and *value* represents the value to replace. If you omit *value*, *text* replaces any zeros in the data. Use the *O* modifier with *F*, *I*, and *G* phrases. Be sure to provide space in the field width for the decoration text.

```
      'O100<CNOTE> Q< DR> F9.2' ⎕FMT E
¯65423.43
   ¯10.00
    ¯0.40
  0.00 DR
    CNOTE
```

If you include multiple *O* modifiers in the same format phrase, *⎕FMT* uses only the last. If you also specify the *L* modifier (for *F* or *I* phrases only), *⎕FMT* left justifies the text.

```
      'O<NONE> LQ< DR> F11.2' ⎕FMT E
¯65423.43
¯10.00
¯0.40
NONE
100.00 DR
```

P — Positive Left Decoration

The *P* modifier places text that you specify to the immediate left of an edited positive or 0 value. It takes the form

P<text>

where *text* represents the text. Use the *P* modifier with *F*, *I*, and *G* phrases. Be sure to provide space in the field width for the decoration text.

```
      'P<+>F10.1' ⎕FMT E
 ¯65423.4
     ¯10.0
      ¯0.4
      +0.0
    +100.0
```

Q — Positive Right Decoration

The *Q* modifier places text that you specify to the immediate right of an edited positive or 0 value It takes the form

Q<text>

where *text* represents the text. Use the *Q* modifier with *F*, *I*, and *G* phrases. Be sure to provide space in the field width for the decoration text.

```
      'Q< DR>I10' ⎕FMT E
    ¯65423
       ¯10
      0 DR
      0 DR
    100 DR
```

When you use the *M*, *N*, *O*, *P*, and *Q* modifiers with the *G* format phrase, the decoration text supersedes text characters in the pattern. Because the decoration text appears adjacent to leading or trailing nonzero digits in the result, the text phrases may not align.

```
      FMT ← 'M<>N< CR>O<NONE> Q< DR>
G<Z,ZZZ--->'
      FMT ⎕FMT ¯123 120 242 0 ¯100 1000
  123 CR
  12 DR-
  242 DR
   NONE
  1 CR--
1 DR ---
```

R — Background Fill

The *R* modifier fills the result field with text that you specify in all of the positions not filled with the edited value. It takes the form

R<text>

where *text* represents the text. Starting at the left side of the field, *text* is repeated as many times as necessary to fill the field. Use the *R* modifier with *A*, *E*, *F*, *I*, and *G* phrases. Be sure to provide space in the field width for the decoration text.

```
      'R<• >I10' ⎕FMT EX
• • ¯65423
• • • •¯10
• • • • •0
• • • • •0
• • • •100

      'R<BACKGROUND>I21' ⎕FMT EX
BACKGROUNDBACKG¯65423
BACKGROUNDBACKGROU¯10
BACKGROUNDBACKGROUND0
BACKGROUNDBACKGROUND0
BACKGROUNDBACKGROU100
```

When used with the *G* format phrase, the *R* modifier is displayed only in positions not occupied by text characters or decorations.

```
      RDEC ← 'R<*>G<$ ZZZ,ZZZ,ZZ9>'
      RDEC ⎕FMT 23987458 38794 287 0
$ *23,987,458
$ *****38,794
$ ********287
$ **********0
```

You can use the *R* modifier to replace leading blanks in the result of an *A* format phrase with nonblank characters.

```
      'R<->A2' ⎕FMT 3 3ρ'BOAURNSEQ'
-B-O-A
-U-R-N
-S-E-Q
```

The *R* modifier does not put the background text into a data position; rather, it only fills in leading positions created by using a repetition factor.

```
      'R<*>A2' ⎕FMT 1 8 ρ'JOHN DOE'
*J*O*H*N* *D*O*E
```

The blank between *JOHN* and *DOE* is not filled in with a background star.

S — Standard Symbol Substitution

The *S* modifier substitutes symbols of your choice for standard symbols used by ⎕*FMT*. Use the *S* modifier with *F*, *G*, and *I* format phrases. You can also use it with the *G* format phrase to free the digit selectors 9 and Z to serve as characters in the result, and with an *F* or *I* format phrase to tailor other formatting effects to individual needs. The *S* modifier allows only a one-to-one substitution of symbols.

The *S* modifier takes the form

S<symbolpairs>

where the first symbol in each pair must be one of the symbols in Table 4-5, and the second symbol is the temporary replacement for the first.

Table 4-5 lists the symbols that you can replace with the *S* modifier and the applicable format phrases for each.

Table 4-5. Valid Symbols

Symbol	Format Phrases	Purpose
9	*G*	Digit selector
Z	*G*	Digit selector; suppresses leading and trailing 0s
*	*E F G I*	Field overflow fill character
.	*F*	Decimal point
,	*F I*	*C* modifier insert character
0	*F G I*	*Z* modifier fill character or leading-zero fill character from a 9 digit selector
-	*F G I*	Position marker used after exhausting significant digits

The following example shows how you can use the *S* modifier to display decimal numbers as clock times.

```
      'S<.:>F7.2' ⎕FMT 1 4 ρ10+0.15 × ¯1+ι4
10:00  10:15  10:30  10:45
```

The substitution affects only the format phrase with the *S* modifier, as in the next example.

```
     'S<.V>F5.1,F9.3'⎕FMT 1 2ρ470.6 370.168
470V6 370.168
```

You can substitute for more than one symbol in a single format phrase. For example, you can use symbol substitution to follow European conventions, where periods rather than commas are used as number separators, and commas rather than periods are used as decimal points to mark the fractional part of a value.

```
      'S<.,,.>CF14.2' ⎕FMT 1234567.89
1.234.567,89
```

The only substitutions permitted for symbols in the left argument to *⎕FMT* are those for the digit selectors 9 and *Z* in the *G* format phrase. The other substitutions that are permitted affect symbols that *⎕FMT* places in the result. In the next example, the *S* modifier frees the digit selector 9 for use as a character in *text*.

```
      'S<9_>G<19__>' ⎕FMT 64 57 72 54
1964
1957
1972
1954
```

Use the *S* modifier to replace the stars that mark field overflow.

```
      A ← ○ 1E2 1
      'F5.2' ⎕FMT A
*****
 3.14

      'S<*$>F5.2' ⎕FMT A
$$$$$
 3.14
```

The following uses of standard symbol substitution produce a *FORMAT ERROR*:

- an odd number of symbols appears between the delimiters; for example, *S*<.>
- the first symbol in a pair of symbols is not in the preceding table; for example, *S*<*V*.>
- more than one substitution is made for the same symbol in the same format phrase; for example, *S*<.:.,>

- the same symbol is substituted for the digit selectors 9 and Z; for example, *S*<9_Z_>.

Z — Zero Fill

The *Z* modifier fills unused leading positions in the result field with 0s instead of blanks. Use the *Z* modifier with *I* and *F* format phrases.

```
      'ZI10' ⎕FMT 3×1 2 3 4 5
0000000003
0000000006
0000000003
0000000012
0000000015
```

You can also use the Z modifier to display telephone numbers when exchanges and line numbers are stored separately.

```
      EXCH ← 355 298 385 448
      NOS ← 56 7980 230 66
      'I3,<->,ZI4' ⎕FMT EXCH NOS
355-0056
298-7980
385-0230
448-0066
```

When you use the *Z* modifier with an editing phrase to format a negative value, the negative sign is left-justified in the result field.

```
      'ZI5' ⎕FMT ¯1 0 1
¯0001
00000
00001
```

Combining Modifiers

When you use the *B* modifier and *O*, *P*, or *Q* modifiers together, any data values that are 0 appear as blanks, and the decoration text does not appear in the result field.

```
      'BO<NONE>I5' ⎕FMT ¯1 0 1
   ¯1

    1

      'BP<+ >I5' ⎕FMT ¯1 0 1
   ¯1
  + 1
```

When you use the *B* and *R* modifiers together, any data values that are 0 do not suppress the *R* fill text in the result field.

```
      'BR< NONE>F5.2' ⎕FMT ¯1 0 1
¯1.00
 NONE
 1.00
```

When you use the *B* and *Z* modifiers together, any data values that are 0 appear as blanks in the result field.

```
      'BZI5' ⎕FMT ¯1 0 1
¯0001

00001
```

When you use the *L* and *O* modifiers together, the text specified by the *O* modifier is left-justified.

```
      'LO<NONE>I6' ⎕FMT ¯1 0 1
¯1
NONE
1
```

When you use the *L* and *Z* modifiers together, the *Z* modifier has no effect.

```
      'LZI5' ⎕FMT ¯1 0 1
¯1
0
1
```

When you use the *M* and *Z* modifiers together, each negative data value appears with its negative sign replaced by the *M* decoration text. The decoration text is left-justified in the result field.

```
      'M<- >ZI5' ⎕FMT ¯1 0 1
- 001
00000
00001
```

When you use the *O* modifier and the *P* or *Q* modifier together without the *B* modifier, any data values that are 0 are formatted by the *O* text rather than the *P* or *Q* text.

```
      'O<    ∇>P<[>Q<]>LI5' ⎕FMT 0 1 2
    ∇
[1]
[2]
```

When you use the *P* and *Z* modifiers together, each 0 or positive data value appears with the *P* decoration text and is left-justified in the result field.

```
      'P<+ >ZI5' ⎕FMT ¯1 0 1
¯0001
+ 000
+ 001
```

When you use the *R* modifier with one or more *M*, *N*, *O*, *P*, or *Q* modifiers, the decoration text overrides the background text for the portion of the field covered by the *M*, *N*, *O*, *P*, or *Q* modifier.

```
      FS←'R<*>M<->N<->O<NONE>P<+>
Q<+>I5'
      FS ⎕FMT ¯1 0 1
**-1-
*NONE
**+1+
```

Stars or Underscores in Result

Some formatting errors cause stars or underscores in your result. This section explains what causes stars or underscores in a result and how you can avoid them.

Stars in Result

If stars appear in your result when you did not expect them, you probably used ⎕*FMT* incorrectly. Stars appear in the result when:

- a formatted value is larger than the field width (often because you forgot to account for the width of decorations in the field width)
- you used an *A* format phrase to edit numeric data
- you used a format phrase other than *A* to edit character data.

To avoid these conditions, use a larger field width, scale the data, or by use the correct format phrase for the datatype.

You can also substitute another symbol for the star by using the *S* modifier (see the section "Modifiers" in this chapter).

Underscores in Result

The precision of the result of ⎕*FMT* is independent of the value of ⎕*PP*. ⎕*FMT* displays up to 18 digits. A format phrase that requests more significant digits than are available in the internal representation uses underscores for missing digits.

```
      'F24.1' ⎕FMT ○1E20
 314159265358979312___._
```

Useful Applications

This section contains examples of commercial applications that use *⎕FMT* with combinations of modifiers. The numeric array ***NUM*** is used in most of the examples.

```
      NUM
1316026.715    755715.3407  4587093.116
  11586.9     2190044.457   ¯471009.5613
6789137.817         0       9347189.744
3836077.871  ¯5195105.458       234.5
```

- Float Dollar Signs and Place Negative Values in Parentheses

```
      'C P<$>Q< >M<($> N<)> F19.2' ⎕FMT NUM
$1,316,026.72      $755,175.34
$4,587,093.12
   $11,586.90    $2,190,044.46
($471,009.56)
$6,789,137.82            $0.00
$9,347,189.74
$3,836,077.87   ($5,195,105.46)
$234.50
```

- Format Negative Values into Separate Columns and Omit Negative Signs

```
'M<>N<                > CI40' ⎕FMT ,NUM

          1,316,027
                      755,715
                    4,587,093
                       11,587
                    2,190,044
      471,010
                    6,789,138
                            0
                    9,347,190
                    3,836,078
    5,195,105
                          235
```

■ **Display Decorations Only**

```
        BOOLEAN ← 0 0 1 0 1
        'P<YES>R<NO  >BI4, X-1,
< >' ⎕FMT BOOLEAN
NO
NO
YES
NO
YES
```

Using the *FORMAT* Workspace

The *FORMAT* workspace contains several functions that help you place titles and row and column names on a report. These functions fall into the following categories:

- title functions that position titles over the report
- label functions that label rows or columns of the report
- a special function that builds custom titling routines.

Use these functions in conjunction with ⎕*FMT* to help you generate reports. Instead of loading the entire workspace, you can copy any of these functions into your workspace with the command

```
      )COPY  FORMAT objects
```

where *objects* is a list of one or more desired functions.

For details and examples of the syntax of the functions, see the Generating Reports chapter in the *Utilities Manual*.

5

Managing Screen and Keyboard Data

This chapter introduces the APL functions for screen and keyboard management in APL★PLUS II/386.

- "Simple Input and Output Management Facilities" explains the inherent input and output capabilities of APL.
- "Manipulating Keystrokes" describes some useful system functions for capturing and manipulating keystrokes.
- "Controlling Screen Operations" discusses the basic screen management facilities provided in the system.
- "Recording and Playing Back Keystrokes" describes some useful system functions for capturing, manipulating, and playing back keystrokes and APL sessions.
- "Using Color" describes how to select and use colors, provided that you have a color monitor and an appropriate color graphics card.

Simple Input and Output Management Facilities

The building blocks of simple input and output facilities are the quad (⎕) for numeric data and quote-quad (⍞) for character data. You can use the quad in an expression like ⎕←*A* to display the contents of *A*. You can also use quad (⎕) in the expression *A*←⎕ to wait for input and store it in the variable *A*. The quote-quad (⍞) works the same way for character input, except that on output (⍞←*A*) it does not display a carriage return at the end of the data.

A typical function might use both quad (⎕) or quote-quad (⍞) to accept input data for processing and to display the result of the operation:

```
     ∇UPDATE
[1]  ⎕←'ENTER NAME: '
[2]  NAME←⍞
[3]  ⍞←'OK, ',NAME,', HOW OLD ARE YOU? '
[4]  AGE←⍞
     ∇
```

The function above begins execution as follows (your input in bold):

```
      UPDATE
ENTER NAME:
SMITH
OK, SMITH, HOW OLD ARE YOU? 34
```

In the *UPDATE* function, *AGE* includes leading spaces to indicate the location of the entry on the line. Some older APL★PLUS systems return the entire line (including the prompt) rather than spaces. If you have programs that require this behavior, you can set ⎕*PR* to '' to emulate the older style. See ⎕*PR* in the System Functions, Variables, and Constants chapter of the *Reference Manual* for details.

You can also use the expression

```
        ⍞←'PROMPT' ◊ ⎕ARBOUT ⍳0 ◊ ANSWER←⍞
```

to eliminate anything on the line except the user's entry. The expression ⎕*ARBOUT* ⍳0 is a special definition of ⎕*ARBOUT*. (Normally ⎕*ARBOUT* is used to send any ASCII character to the screen. It sends arbitrary character data to the screen and interprets the numeric codes given in its argument as the corresponding ASCII character.) After you use ⎕*ARBOUT* ⍳0 to eliminate the prompt, you cannot backspace into the prompt or move the cursor off the input line.

Using Terminal Control Characters

APL★PLUS II/386 provides several system constants to simplify sending control characters to the screen or to the printer. You can use them to produce special actions. The system constants take the form ⎕*TCnn* and produce the following control characters:

• Bell	⎕*TCBEL* (system constant)
• Backspace	⎕*TCBS* (system constant)
• Delete	⎕*TCDEL* (system constant)
• Escape	⎕*TCESC* (system constant)
• Form Feed	⎕*TCFF* (system constant)
• Linefeed	⎕*TCLF* (system constant)
• Newline	⎕*TCNL* (system constant)
• Null	⎕*TCNUL* (system constant)
• Tab	⎕*TCHT* (system constant)

Note how some of these constants are used in the example in the following section. For details on the terminal control system constants, see ⎕*TCxx* in the System Functions, Variables, and Constants chapter of the *Reference Manual*.

Placing Output on the Screen

In conjunction with quad (⎕) and quote-quad (⍞) on output, the session-related system variable ⎕*CURSOR* can position the output anywhere on the screen. The session-related variable ⎕*CURSOR* contains the current location of the cursor within the current window on the screen. Its value is a vector that contains the row and column of the current cursor position relative to the upper left corner of the window in use (see ⎕*WINDOW* in the section "Controlling Screen Operations" in this chapter).

You can assign ⎕*CURSOR* a new value, which will immediately cause the cursor to move to a new location. The following function, a slightly enhanced *UPDATE* program, clears the screen and positions the question on the screen:

```
     ∇UPDATE2
[1]   ⎕TCFF
[2]   ⎕CURSOR ← 4 15
[3]   ⍞←'ENTER NAME: ' ⋄ ⎕ARBOUT ⍳0
[4]   NAME←⍞
[5]   REP←'OK, ',NAME,', HOW OLD ARE YOU? '
[6]   ⎕CURSOR ← 6 10
[7]   ⍞←REP,⎕TCBEL ⋄ ⎕ARBOUT ⍳0
[8]   AGE←⍞
     ∇
```

Manipulating Keystrokes

The system has two mechanisms for manipulating keystrokes ⎕*INKEY* and ⎕*INBUF*.

- ⎕*INKEY* captures one keystroke at a time
- ⎕*INBUF* stores keystrokes in the event buffer.

You can also use the system function ⎕*PFKEY* to customize the programmable function keys on your computer.

This section explains what the event buffer is and how you use ⎕*INKEY* and ⎕*INBUF* with it. It also describes how you use ⎕*PFKEY* to customize function keys for use in your applications. Other mechanisms for manipulating keystrokes are discussed in the "Recording and Replaying APL Sessions" section later in this chapter.

What Are the Type-Ahead and Event Buffers?

The system provides a storage area for anything that you type called the **type-ahead buffer**. This storage area allows you to continue to type while the system is busy processing previous instructions. What you type ahead is held in the type-ahead buffer until the system is ready to process it. For example, you could enter 6⍴⎕*TCBEL* during a long operation. When the system completes the operation, it executes 6⍴⎕*TCBEL*. This expression makes the computer beep, which tells you that the operation is complete.

You can enter up to 256 keystrokes into the type-ahead buffer. The system acts on some keystrokes, like Pause or Break, immediately, without placing them in the type-ahead buffer.

When the system needs keyboard input, it looks in the **event buffer**. You can place events (representing both characters and keyboard actions) in the event buffer with the system function ⎕*INBUF*. If the event buffer is empty, ⎕*INBUF* converts a keystroke in the type-ahead buffer to an event.

The system discards any unused portion of the contents of the type-ahead buffer when you use Exit or Break, or through a special use of ⎕*INBUF*. The type-ahead buffer is not cleared when an APL error occurs. Therefore, if an error halts a program and the type-ahead buffer contains unused material, the system uses it as if you had just typed it at the keyboard in response to the error halt.

APL★PLUS II/386 also executes system commands from the buffer, but there are distinct dangers in using them under program control because the system cannot trap errors from system commands. Nearly everything that you can do with a system command in APL★PLUS II/386 (including managing saved workspaces), you can do more safely with a system function.

You can simulate pressing keys by using the system function ⎕*INBUF*. Keystrokes that you simulate with ⎕*INBUF* are not actually placed into the 256-keystroke type-ahead buffer; rather, they are placed into the event buffer whose size you can specify with the `Holdbuf=` startup parameter. Conceptually, the system treats these keystrokes as though you physically typed them ahead, although the event buffer can also contain keyboard actions that you cannot actually type from the keyboard (see the Advanced Techniques chapter).

Capturing Single Keystrokes

The system function ⎕*INKEY* allows you to write APL programs that react to each keystroke as it is typed.

To capture a single keystroke, you use ⎕*INKEY*. While quote-quad (⍞) captures all input until you press Enter, ⎕*INKEY* takes only a single input keystroke. For example:

```
⍞←'PRESS ANY KEY TO CONTINUE' ⋄ KEYSTROKE←⎕INKEY
```

Storing Keystrokes in the Event Buffer

The system function ⎕*INBUF* can simulate typing; the simulated keystrokes can be placed either before or after keystrokes waiting in the event buffer. ⎕*INBUF* requires a right argument; a left argument is optional.

The right argument is a vector of keystrokes to be stored. If all of the keystrokes are elements of ⎕*AV*, the vector can be character. A numeric vector allows both characters and other keystrokes, such as cursor movements, using the numeric event codes listed in the Atomic Vector and Keyboard Events appendix in the *Reference Manual*.

The optional left argument specifies where you place the keystrokes in the event buffer:

- 1 — insert before the current contents of the buffer (default)
- 0 — replace the current contents of the buffer
- ¯1 — insert after the current contents of the buffer.

Consider using ⎕*INBUF* to clear the event buffer when errors that can be trapped occur. (See ⎕*ELX* in the *Reference Manual*.)

You can use ⎕*INBUF* to clear the event buffer of accidental input. The statement

```
      0 ⎕INBUF ''
```

discards the current contents of the buffer and leaves it empty.

You can also use ⎕*INBUF* to place a default answer in an input field for a user to accept or change; for example:

```
      1 ⎕INBUF 'DEFAULT ANSWER' ◊ A←⍞
```

Customizing PFkeys

Most personal computer keyboards have a set of function keys or PFkeys. You can set these keys to have special meanings. The system function ⎕*PFKEY* allows you to set the PFkeys to anything you want in the APL environment.

Reading and Setting PFkeys

You have 999 function key numbers available for your use. By default, the keyboard driver associates the logical PFkey numbers and physical keystrokes shown in Table 5-1.

Table 5-1. Logical PFkeys and Physical Keystrokes

Key Number	Physical Keystroke
1 through 12	F1 through F12
101 through 112	Shift-F1 through Shift-F12
201 through 212	Ctrl-F1 through Ctrl-F12

You can change this association or assign other PFkey numbers to other keystrokes by modifying the keyboard tables. The Advanced Techniques chapter explains how you modify the tables; the Atomic Vector and Keyboard Events appendix of the *Reference Manual* lists the event numbers.

The ⎕*PFKEY* function requires a right argument and takes an optional left argument. The right argument is an integer that represents the function key whose contents you want to examine or change.

To find out which keystrokes are assigned to a PFkey, use ⎕*PFKEY* monadically. The following example shows that PFkey F1 has been assigned the APL system command)*VARS*.

```
      ⎕PFKEY 1
)VARS
```

If you use a negative integer for monadic ⎕*PFKEY*, the explicit result is a fully numeric form that represents keystroke values. These values allow you to recognize invisible characters like Space and ⎕*TCNUL*. For example, to see the keystroke values assigned to PFkey 1, you enter:

```
      ⎕PFKEY ¯1
41 86 65 82 83 13
```

To reset F1 to the system command)*FNS*, supply the system command, enclosed in single quotes, as the left argument to ⎕*PFKEY*

```
      ')FNS' ⎕PFKEY 1
      ⎕PFKEY 1
)FNS
```

You can see how setting PFkeys to commonly used keystroke sequences can save typing. Instead of typing the same series of commands repeatedly, you can assign them to a key and type the commands with one keystroke. For example, if you always want to clear the state indicator before saving a workspace, you can set a PFkey with

```
      (')SIC',⎕TCNL,')SAVE',⎕TCNL) ⎕PFKEY 2
```

When you press F2, you perform the)*SIC* and)*SAVE* commands with one keystroke.

The ⎕*PFKEYS* function allows you to store all of your PFkey settings in one localizable array. Until you set a key, it has no value and no effect when you press it. PFkey settings remain in effect only through a current session; they are lost when you sign off. You can preset PFkeys in an initial workspace or in the configuration file. See ⎕*PFKEY* and ⎕*PFKEYS* in the System Functions, Variables, and Constants chapter of the *Reference Manual* for more information.

You can also set PFkeys interactively in the session manager. Press Shift-Esc, then press the function key you want to define. The status line changes to allow you to type the contents of the new function key. If you make a mistake while typing, press Alt-Shift-Backspace to erase the mistake. To finish the definition, press the function key again.

Setting PFkeys in Applications

You generally want your users to be able to use your applications easily. One method is allow the user to press a PFkey to execute a particular action. For example, F1 could invoke a Help facility or F10 could exit the application. Since many applications determine keystrokes with *⎕INKEY*, you can store one "key" on a PFkey and have your program recognize it. For the preceding example, you could have your program execute:

```
        ⎕AV[210] ⎕PFKEY 1
        ⎕AV[212] ⎕PFKEY 10
```

The elements of the left arguments are line drawing characters. Since they are unlikely to be typed from the keyboard, they prevent interference with other keystrokes. Your program could check for these keys with a statement like:

```
        →(210 212=⎕AV⍳KEY←⎕INKEY)/HELP, EXIT
```

The system checks the keystroke accepted by *⎕INKEY* to see if one of the PFkeys was pressed, and the program branches appropriately if true.

Recording and Replaying APL Sessions

There are three system variables, ⎕*KEYLOG*, ⎕*KEYSRC*, and ⎕*LINELOG*, and three startup parameters, `keylog`, `keysrc`, and `linelog`, that enable you to record and play back APL sessions.

- ⎕*KEYLOG* records keystrokes into a native file. You can use this file with ⎕*KEYSRC* to replay session activity.
- ⎕*KEYSRC* executes keystrokes from a specified native file rather than from the keyboard.
- ⎕*LINELOG* records lines of input and output from an APL session into a file.

Recording Session Input

The ⎕*KEYLOG* function records physical keystrokes into a native file, two bytes per keystroke. To begin recording keystrokes, enter

```
      ⎕KEYLOG←'filename  R'
```

where *filename* is the name of the file you want to record keystrokes in. If the file you specify already exists, the system adds the keystrokes you type to the end of the file. If the file does not exist, the system creates it.

R is an optional argument that replaces the contents of an existing file with the keystrokes you enter.

When the system is recording, the letters "Log" appear in the status line. (If the workspace pathname is more than 25 characters long, these letters may be overwritten.) To stop recording, enter:

```
      ⎕KEYLOG←''
```

The letters "Log" no longer appear in the status line. To view or edit the contents of a file created using *⎕KEYLOG*, use the *KEYEDIT* workspace. See "Using the *KEYEDIT* Workspace" in this chapter for details.

If an error — such as Disk Full — occurs while you are recording keystrokes, a bell sounds and the message "Keyboard log file disk error; press Enter" appears in the status line. *⎕KEYLOG* is reset to an empty vector.

You can record keystrokes in only one file at a time. If you assign a new filename to *⎕KEYLOG* during a recording session, the system stops recording keystrokes in the current keystroke file and begins recording them in the new file. To display the name of the current keystroke file, enter

```
      ⎕KEYLOG
```

⎕KEYLOG records only keystrokes that you type, not keys placed in the event buffer by *⎕INBUF*, superevents, or mouse activity. PFkeys are not expanded. PFkey values (601 to 1600) are recorded in the file — not their current contents. The following keystrokes and events have special uses in *⎕KEYLOG*.

- **Ctrl-1**
  This keystroke inserts event 337 in the current *⎕KEYLOG* file. When you replay this file using *⎕KEYSRC*, the system temporarily reverts to keyboard input when it encounters event 337. Keystroke playback resumes when you press the Enter key.

- **Ctrl-2**
  This keystroke inserts event 338 in the current ⎕*KEYLOG* file. When you replay this file using ⎕*KEYSRC*, the system permanently stops reading the keystroke file when it encounters event 338 in the file and input reverts to the keyboard.

- **Ctrl-3 (Event 339)**
  This keystroke temporarily suspends keystroke capture. The cursor becomes a block cursor to remind you of the suspension and the letters "Log" disappear from the status line. To resume keystroke capture, press Ctrl-3 again. Suspending keystroke capture does not affect the the contents of ⎕*KEYLOG* and the Ctrl-3 keystrokes are not recorded in the keystroke file.

- **Event 291**
  When you place this event in the input file, ⎕*KEYSRC* delays execution of the following keystroke for one second. You can assign this event to a keystroke combination and insert it in the input file wherever you want a one second delay. To assign the event to a keystroke combination, use the `key n =` startup parameter.

  You can also use the *KEYEDIT* workspace to insert this event after you finish recording keystrokes. See the "Editing Keystroke Files" section in this chapter for information on editing keystroke files.

Replaying Keystrokes

The *⎕KEYSRC* function obtains keystrokes from a native file rather than from the keyboard. You can use it to replay keystrokes captured using *⎕KEYLOG*. The system obtains a keystroke from the file only when it needs a keystroke from the keyboard (that is, when *⎕INBUF* is empty). To replay a *⎕KEYLOG* file, enter

```
    ⎕KEYSRC←'filename  n'
```

where *filename* is the name of the *⎕KEYLOG* file you want to replay. If the file does not exist or cannot be tied, the system displays an error message.

n is an even non-negative integer specifying the offset into the file at which the system begins to read the file. If you do not specify a number, the system reads the keystrokes from the beginning of the file.

The *⎕KEYSRC* function replays the keystroke file until it reaches the end-of-file. Input then reverts to the keyboard and *⎕KEYSRC* is reset to an empty vector. If an error occurs when reading the keystroke file, a bell sounds and the message "Keyboard source file disk error; press Enter" appears in the status line. *⎕KEYSRC* is reset to an empty vector.

The system always recognizes superevents typed at the keyboard. For example, if you press Break or Exit (Ctrl-Esc) while the system is reading from a keystroke file, input reverts to the keyboard and *⎕KEYSRC* is reset to an empty vector. You can type ordinary keystrokes while *⎕KEYSRC* is replaying a keystroke file; however, the system does not execute those keystrokes until input reverts to the keyboard.

If *⎕KEYSRC* finds the following keystrokes in the keystroke file, it interprets them as described below.

- **Ctrl-1 (event 337)** These keystrokes in the input file cause input to revert temporarily to the keyboard. Input from the keystroke file resumes when you press the Enter key.

- **Ctrl-2 (event 338)** These keystrokes in the input file cause input to revert to the physical keyboard and reset *⎕KEYSRC* to an empty vector. The system ignores any additional keystrokes in the input file.

- **Event 291** This event causes the system to delay for 1-second. You can assign this event to a keystroke combination and insert it in the input file wherever you want a 1 second delay. To assign the event to a keystroke combination, use the `key n =` startup parameter.

  You can also edit the input file using the *KEYEDIT* workspace and insert this event after the system records the keystrokes. See the "Editing Keystroke Files" section in this chapter for information on editing keystroke files.

Creating a Log of Session Input and Output

You can record the input and output from an APL session using *⎕LINELOG*. To begin recording session activity, enter

```
        ⎕LINELOG←'filename  R'
```

where *filename* is the name of the file you want to record session activity in. If the file you specify already exists, the system adds any lines that you type to the end of the file. If the file you specify does not exist, the system creates it.

R is an optional argument. When you use this argument, the system replaces the contents of the file you specify.

⎕LINELOG writes all lines of input and output to an ordinary text file with a carriage return and linefeed between each line. However, *⎕LINELOG* does not write an explicit end-of-file marker at the end of the file.

While the system is recording, the letters "Log" appear in the status line. (If the workspace pathname is more than 25 characters long, these letters may be overwritten.) To stop recording lines, enter:

```
        ⎕LINELOG←''
```

You can view the contents of the log file by reading it into any text editor or into the session manager editor. If the log file is long, you may not be able to load the entire log file into the session manager editor. To view exceptionally long files, use the browse mode of the editor.

If an error — such as Disk Full — occurs while you are recording lines, a bell sounds and the message "Line log file disk error; press Enter" appears in the status line. *⎕LINELOG* is reset to an empty vector, and the system waits for the next keystroke.

You can record lines in only one *⎕LINELOG* file at a time. If you assign a new filename to *⎕LINELOG* during a APL session, the system stops recording lines in the current *⎕LINELOG* file and begins recording lines in the new file. To display the name of the current *⎕LINELOG* file, enter:

```
        ⎕LINELOG
```

The *⎕LINELOG* function cannot record activity in editing sessions, material placed on the screen by *⎕WPUT*, or editing activities — such as copying or moving blocks. It also does not take into account the ability to move the cursor back into previously executed lines. If this occurs, the contents will not always match the lines in the scrolling APL session. To preserve the contents of the scrolling APL session, tag the lines, save them in a variable with Ctrl-V, and then write the variable to a file. The lines that *⎕LINELOG* records into the file are the same as the lines that would be sent to the printer if you turned on printer-slaving with Ctrl-PrtScr.

Recording and Replaying APL Sessions Using Startup Parameters

The following three startup parameters allow you to record and replay APL sessions. You can include them either in the configuration file or on the APL command line.

- **`keylog`**
  Use this parameter to record keystrokes from the APL session in a file that you specify. The keylog startup parameter specifies the initial value for ⎕*KEYLOG*. To record keystrokes in a file, enter `keylog` *filename*. To replace the contents of an existing file, use the R parameter; for example, `keylog='`*filename* `R'`. If you use the parameter, be sure to enclose the filename and parameter in quotes.

- **`keysrc`**
  Use this parameter to replay the keystrokes contained in a file that you specify. This startup parameter specifies the initial value for ⎕*KEYSRC*. To replay a keystrokes file, enter `keysrc=`*filename*.

  You can specify an even offset number to indicate where in the file you want the system to begin replaying keystrokes; for example, `keysrc='`*filename* `100'`. If you use an offset number, be sure to enclose the filename and offset number in quotes. If the system cannot tie the file you specify, it ends the APL session.

- **`linelog`**
  Use this parameter to record all input and output from the APL session in a file that you specify. This startup parameter specifies the initial value for ⎕*LINELOG*. To record an APL session in a file, enter `linelog=`*filename*. To replace the contents of an existing file, use the R parameter; for example, `linelog='`*filename* `R'`. If you use the parameter, be sure to enclose the filename and parameter in quotes.

Using the *KEYEDIT* Workspace

The *KEYEDIT* workspace contains functions that let you view and edit keystroke files created using ⎕*KEYLOG*. This workspace requires the startup parameters `PFNUM=200` and `PFMEM=10000`. You cannot use this workspace without setting these parameters.

The functions in the *KEYEDIT* workspace are described below.

- *CREATEREP*
  Examines the keyboard table and creates global variables that contain representations of the keyboard and PFkeys. The *KEYEDIT* workspace comes with default global variables, and you do not have to run *CREATEREP* unless you change the default keyboard with the `key=` startup parameter or the *KEYBOARD* workspace.
- *KEYEDIT*
  Lets you edit a keystrokes file.
- *KEYDELAY*
  Inserts delays after each keystroke in the file.
- *LINEDELAY*
  Inserts delays after each line in a file.
- *UNDELAY*
  Removes delays from a file.

Editing Keystroke Files

The *KEYEDIT* function lets you edit a keystroke file created by ⎕*KEYLOG*. To edit a keystroke file, enter:

```
        KEYEDIT 'filename'
```

If you do not include a period with the filename, *KEYEDIT* adds a default extension .KST.

Two states in the *KEYEDIT* function — Meta and Keystroke — determine what effect the keys you press have during a *KEYEDIT* session. The status line identifies the state you are in. To toggle between the two states, press the Esc key.

When you press a key in the Meta state, the key performs the action associated with it. For example, when you press the A key the system displays the letter "A" on the screen; when you press the Delete key, the system deletes the character at the cursor position. In the Keystroke state, pressing a key generates a symbolic representation of that key on the screen. For example, pressing the Delete key causes the letters "<Del>" to appear on the screen rather than deleting a character in the keystroke file. When you replay the keystroke file using *⎕KEYSRC*, the system performs the action associated with the key you pressed.

To end a *KEYEDIT* session, press the Esc key to switch to the Meta state and then press either Ctrl-E or Ctrl-Q. Ctrl-E lets you save any changes to the keystrokes file; Ctrl-Q ends the session without saving changes.

Controlling Screen Operations

The APL★PLUS II/386 System has one system variable and two system functions that let you perform basic screen operations.

- `⎕WINDOW` defines the size of the display window on the screen
- `⎕WPUT` places data on the screen exactly where you want it
- `⎕WGET` retrieves data from the screen and stores it in a variable.

Defining Windows

The system variable `⎕WINDOW` defines the current window in a session. The default value is `0 0 25 80` or the setting determined by your hardware video mode, which positions the anchor at the top left corner of the display screen. The last two numbers define the size of the window in character positions; the default is the size of your monitor's display, normally 25 rows by 80 columns. Notice that the starting value for the row and columns is 0, which means that the last row number is 24 and the last column number is 79 for the default screen. You can change the current window at any time by re-assigning it to a new value. For example, executing

```
      ⎕WINDOW←1 0 24 80
```

will not include the top line of your monitor's display in the current window. Try assigning different values to `⎕WINDOW` and moving the cursor around the screen to see the effects. If you make the size too large for the anchor position, you receive a *DOMAIN ERROR*.

```
      ⎕WINDOW←10 0 24 80
```

To find the size of the current window, use `⎕WINDOW` without assigning it.

Displaying Data on the Screen

If you have a large quantity of text to display or if you want to display different attributes, ⎕*WPUT* is a useful tool. This system function replaces the characters or attributes (or both) in a specified screen window. ⎕*WPUT* has the syntax:

```
      wspec ⎕WPUT data
```

The optional left argument specifies the position on the screen for the data you want displayed; the default is the current value of ⎕*WINDOW*.

The right argument can contain character or numeric data. The ⎕*WPUT* function shapes the data to fit the size of the window. If the data are too small, ⎕*WPUT* reuses it until the window is full. If the data are too large, ⎕*WPUT* uses only what it needs. If the right argument is a character scalar, vector, or matrix, ⎕*WPUT* displays those characters in the current window or in the windows specified in the left argument. Try

```
      ⎕WPUT 'ABC'
```

or

```
      10 10 10 10 ⎕WPUT 'B'
```

and notice the results.

You can also use the display attributes of your computer in conjunction with ⎕*WPUT*. If the right argument is a numeric scalar, vector, or matrix, ⎕*WPUT* displays different attributes in the current window. For example, try

```
      ⎕WPUT 112
```

to use reverse video. Table 5-2 shows the IBM-specific attribute values for a monochrome screen. To determine color attribute values, see the "Using Color" section later in this chapter.

Table 5-2. IBM-Specific Attribute Values

Attribute	Description
0	nondisplay
1	underline
7	white character/black background
9	underline, high intensity
15	white character/black background, high intensity
112	reverse video
120	reverse video, high intensity
129	underline, blinking
135	white character/black background, blinking
137	underline, high intensity, blinking
143	white character/black background, high intensity, blinking
240	reverse video, blinking
248	reverse video, high intensity, blinking

For more information on attribute values, see ⎕*WPUT* in the System Functions, Variables, and Constants chapter of the *Reference Manual* or your computer's operating manual.

A three-dimensional character array right argument also displays characters with their attributes. The first column of the argument (*ARG*[; ; ⎕*IO*]) contains the characters; the second contains the attributes encoded as characters by ⎕*AV*[*attribute*+⎕*IO*].

Retrieving Data from the Screen

The system function *⎕WGET* reads the characters or attributes (or both) from a specified screen window. It has the syntax:

```
        result ← wspec ⎕WGET rtype
```

This function "gets" part or all of the current screen and returns its result as a variable. *⎕WGET* has exactly the same left argument as *⎕WPUT*. The right argument is one of the following values.

- 1
  The result is a character matrix that describes the characters on the current screen. Use this value to get user input from fields in a full-screen application (see the Developing Full-Screen Applications chapter).
- 2
  The result is a numeric matrix that describes the attributes for each position on the screen.
- 3
  The result is a three-dimensional character array that includes both characters and attributes. In full-screen applications, you can use this value when you need to save an old screen to redisplay later.

Converting Character Data to Numeric Data

Frequently, you must convert data captured from the screen from character data to numeric data. The system function ⎕*FI* converts character vectors that contain digits to actual numeric vectors. Unlike format (⍎), non-numeric characters in the variable do not cause errors, but return a 0 for the characters. The system function ⎕*VI* verifies the character vector by returning a 1 for numbers and a 0 for characters. Use ⎕*VI* when you want to store user input as numbers. You can use ⎕*WGET* to capture values from a screen, ravel the result (⎕*FI* and ⎕*VI* require scalars and vectors only), and then apply ⎕*FI* or ⎕*VI* to it. For example:

```
      FIELD ← , 0 0 1 16 ⎕WGET 1
      FIELD
ABC 123 1 HELP 3
      ⎕FI FIELD
0 123 1 0 3
      ⎕VI FIELD
0 1 1 0 1
      (⎕VI FIELD)/⎕FI FIELD
123 1 3
```

If the argument to ⎕*FI* or ⎕*VI* is all spaces or empty, the result is an empty numeric vector. To ensure that you always return some number, even if the field is empty, use

```
      1↑(⎕FI ,(anchor,size) ⎕WGET 1),missv
```

where *anchor* and *size* determine the location and size of the field, and *missv* is some variable that is used as a missing value indicator. Note this will only work for single value fields.

Using Color

If you have a color monitor, you can set different colors for the following parts of your system:

- APL session and status line
- edit session and status line
- partially tagged region
- fully tagged region
- wrapped line marker.

You can also set attributes for these parts if you have a monochrome monitor.

You can control whether the high-order bit of an attribute number is interpreted by EGA and VGA hardware to mean "blink the foreground color" or "intensify the background color." The latter allows a wider selection of color combinations.

APL★PLUS II/386 allows you to set these colors in four ways:

- you can use the functions in the supplied *COLOR* workspace to set the colors

- you can use startup parameters to set the colors when you start APL

- you can use ⎕*POKE* to set or change the colors after you start APL or within an application

- you can specify colors for applications that use ⎕*ED*. (See the "Using the Editor in Your Applications" section in the Advanced Techniques chapter for details.)

Using the *COLOR* Workspace

The *COLOR* workspace contains functions that help you determine or set colors. These functions display the possible colors along with the appropriate attribute numbers so that you can see which colors are associated with certain numbers. For a summary of the functions in this workspace, load the workspace and execute:

```
      SUMMARY
```

For additional details on each function, execute:

```
      EXPLAIN 'fnname'
```

Before you can set or change colors, you must know the color attribute number associated with the color you want to use. A color attribute number is an encoded integer ranging from 0 to 255 that designates the foreground color, background color, and intensity of each; for example, 90 designates light green on magenta.

Setting Colors Interactively

The *CBE* function (color-by-example) allows you to select the foreground and background colors interactively. When you use *CBE*, you can see the colors available to you and the color attribute numbers associated with them. Once you select the colors you like, record the color attribute numbers and use them to set your colors permanently.

To use this function, type:

```
      CBE
```

The function causes the system to display a sample APL session and status line, sample edit session and status line, a partially tagged region, a fully tagged region, and a wrapped line marker. The last line of this display shows how you choose the different areas and how you choose the colors.

To select the area you want to color, press one of the following letters:

- *A* — colors the APL session
- *S* — colors the APL session status line
- *E* — colors the edit session
- *L* — colors the edit session status line
- *P* — colors a partially tagged region
- *F* — colors a fully tagged region
- *W* — colors the wrapped line marker.

Use the right, left, up, and down arrow keys (← → ↑ ↓) to cycle through the available colors. The left and right arrow keys cycle through the foreground colors; the up and down arrow keys cycle through the background colors. As you cycle through the colors, the relevant part of the screen displays the color; the color attribute number displays in the upper right corner of the screen. When you finish selecting colors, press Enter to return to the APL session, which uses the new color selections.

You can either note the new color attribute numbers as you change them or retrieve the color attribute numbers using the *SYSATTRS* function.

Determining Colors and Color Attribute Numbers

To see the current colors, use *SHOWSYSATTRS*. This niladic function displays the current colors in this order:

- APL session
- APL session status line
- edit session
- edit session status line
- partially tagged region
- fully tagged region
- wrapped line marker
- border (CGA and VGA only).

To determine the color attribute numbers for the current colors, execute *SYSATTRS*. This niladic function returns the color attribute numbers for the eight regions in the preceding list.

The *ON* and *ATTRNUM* functions display the color attribute numbers for the colors you "describe" to them. You can use the *LIGHT* function in conjunction with *ON* and *ATTRNUM*. For example, to determine the color attribute number for a light blue foreground on a black background, enter:

```
      LIGHT BLUE ON BLACK
9
```

Alternatively, you can use *ATTRNUM* with a character argument to find the color attribute number for a light green foreground on a blue background:

```
      ATTRNUM 'LIGHT GREEN ON BLUE'
26
```

To see colors associated with color attribute numbers, use the *ATTRSPELL* and *SHOWATTRS* functions.

The *ATTRSPELL* function describes the color associated with a single color attribute number:

```
      ATTRSPELL 1
Blue on Black
```

For a visual display, use the *SHOWATTRS* function. This function displays colored blocks across the screen for the attribute numbers you supply. You can also supply a character or characters for *SHOWATTRS* to use in the foreground. (If you do not supply characters, *SHOWATTRS* uses a diamond.) For example, to display the letters *A B C D E* in five colored blocks, using random attributes, enter:

```
      'A B C D E' SHOWATTRS 5?255
```

To see a display of all of the available attributes along with their attribute numbers, execute:

```
      SHOWPANEL ATTRIBUTES
```

Setting and Changing Colors

Once you know the attribute numbers for the colors you want, you can use the *SETATTRS* function to set or change colors quickly. *SETATTRS* is a monadic function that takes a right argument of up to eight attribute numbers that correspond to the eight areas you can color (listed earlier). For example, try

```
      SETATTRS 31 79 63 75 95 91 78
```

to see a sample of colored areas.

The *DARK*, *LIGHT*, *INVERT*, and *REVERSE* functions allow you to manipulate colors. The *DARK* function returns the attribute numbers for darker versions of the attribute numbers you supply. The *LIGHT* function returns the attribute numbers for lighter versions of the attribute numbers you supply. The *INVERT* function inverts the primary color bits as seen by the mouse pointer. The *REVERSE* function returns the attribute numbers for the reverse of the attribute numbers you supply. For example, if you supply the attribute number for a red foreground on a blue background, *REVERSE* returns the attribute number for a blue foreground on a red background.

Setting Colors with Startup Parameters

Four startup parameters allow you to specify colors in the configuration file or on the APL command line.

- `aplattrs`
  This parameter specifies the colors of the APL session and the status line. The defaults are white on black (7) for the APL session and black on white (112) for the status line. The values you specify here also apply to terminal mode.

- `editattrs`
  This parameter specifies the colors of the edit sessions and their status lines. The defaults are white on black (7) for the edit session and black on white (112) for the status line.

- `tagattrs`
  This parameter specifies the colors of a partially tagged region and fully tagged region. The defaults are light white on black (15) for the partially tagged region and light black on white (120) for the fully tagged region.

- `wrapattr`
  This parameter specifies the color you want to use for the wrapped line marker. The default is blue on black (1).

A sample configuration file, COLOR.APL, is included with your system. To use it, place the `config=color.apl` startup parameter in the startup configuration file or include it on the command line.

Setting Colors with ⎕*POKE*

Once you start APL★PLUS II/386, you can change the colors using ⎕*POKE*. You can also set the values in your applications.

Table 5-4 shows the memory locations you can use with ⎕*POKE* to set or change colors.

Warning: Do not poke 0 or 1 into 902 unless you have an EGA, VGA, or other monitor that supports blink switching.

Table 5-4. Values for ⎕POKE to Set or Change Color Attributes

Location	Area to Color	Default	
161	APL session and terminal mode	7	(white on black)
170	APL and terminal mode status line	112	(black on white)
900	Edit session	7	(white on black)
901	Edit session status line	112	(black on white)
899	Partially tagged block/region	15	(light white on black)
889	Fully tagged block/region	120	(light black on white)
120	Wrap attribute	1	(blue on black)
902	Blinking bit (0=off, 1=on, 255=unknown)	255	(unknown)

6 Developing Full-Screen Applications

The ⎕*WIN* system function is the heart of menu and form control applications. Used with ⎕*WKEY*, ⎕*WIN* controls user input on screens and can validate fields. This chapter is organized as follows.

- "Creating and Controlling Screens with ⎕*WIN*" describes the arguments and results of ⎕*WIN* and how you build full-screen input applications.
- "Interpreting Keystrokes with ⎕*WKEY*" explains how you use ⎕*WKEY* to associate actions with keystrokes and define field type values.
- "Advanced Techniques and Utilities" describes tools and techniques that you can use to make your applications more efficient and user friendly.
- "The *WINTUTOR* Workspace" is an overview of the *WINTUTOR* workspace. This workspace contains utility functions and a sample application that can help you learn how to use ⎕*WIN* and ⎕*WKEY*.

The chapter contains examples that you can try; these examples begin with "Try this:" and are shown inside a box. In all of the examples, the left argument to ⎕*WIN* is referred to as *Lwin*, the right argument as *Rwin*, and the result as *Zwin*.

Creating and Controlling Screens with ⎕*WIN*

⎕*WIN* is the heart of screen input control within the APL window. It allows you to define the appearance and placement of fields on a screen, and to specify exit conditions. A field is an area of the screen where a user enters data. An exit condition is an occurrence that causes ⎕*WIN* to stop.

You can use ⎕*WIN* in two major ways:

- you run ⎕*WIN* only once in a function to allow user input
- you run ⎕*WIN* until an exit occurs, you take some action (such as validating the entry or displaying help), then you re-enter ⎕*WIN* transparently. This process continues through the fields.

⎕*WIN* requires a right argument, and optionally uses a left argument. The right argument is a matrix in which each row describes a field. This description includes position, size, type, special features, and display attributes for active and inactive fields. The optional left argument specifies the starting field, the cursor position in that field, the number of seconds the computer waits for input, and the numeric value of the keystroke you want to use.

The result of ⎕*WIN* tells you what kind of exit caused ⎕*WIN* to exit; that is, complete execution and return a result to APL. An exit occurs when you press a certain key or an invalid key. These conditions generally relate to the special features for each field. For example, if you specify the special feature "exit for validation" for a field, ⎕*WIN* exits every time you leave that field so that you can validate the data.

The result also reports which keystroke you pressed, what field you were in, the position of the cursor when the exit occurred, and whether you modifed the field during the last execution.

The Right Argument

The right argument to ⎕*WIN* is a vector (single field) or matrix (multiple fields) that tells the system what to do with each field on a screen. Location, size, valid characters, special actions, and field attributes are all determined by this argument. A field number refers to the row in the matrix that contains the descriptive field information.

Table 6-1 summarizes the meaning of the elements in the right argument.

Table 6-1. Elements of the Right Argument

Element	Meaning
1	Anchor — row in which field begins
2	Anchor — column in which field begins
3	Size — Number of rows in field
4	Size — Number of columns in field
5	Field type — 0, ¯1, or sum of powers of 2
6	Sum of values of special features
7	Display attribute of active field
8	Display attribute of inactive field

The first two columns contain the starting location, or anchor, of a field. The anchor is in origin 0; that is, the upper-left corner of the screen is position 0 0. The size in columns 3 and 4 specify the size of the field by the number of rows and columns. For example, if the field is 1 row and 10 columns placed in position 0 0, the first four values are 0, 0, 1, and 10.

Try this:

Create a one-row, ten-column field in the upper-left corner of your screen:

```
        ⎕TCFF ⋄ ⎕WIN 0 0 1 10
```

The cursor sits in the first column of the field. Type in any 10 letters and numbers. When you enter the next character, the cursor moves back to the beginning of the field and, if the keyboard is in replace mode, overwrites the character already there.

Press Esc to exit from ⎕WIN.

The fifth column contains a field type, which defines the valid keystrokes for the field. All values in this column must be 0, ¯1, or the sum of powers of 2 between 2⋆0 and 2⋆14. ⎕WIN uses these values in conjunction with ⎕WKEY to restrict the characters that a user can type in the field. (See the "Interpreting Keystrokes with ⎕WKEY" section.) If you do not use ⎕WKEY, ⎕WIN uses the following default field types:

- 0 — the user cannot enter any displayable character
- 1 — the user can enter any displayable character (the default)
- 2 — the user can enter the numeric values `0123456789.,-¯E`.

Try this:

Create the same field as in the previous example, but this time restrict valid entries to numeric values.

```
        ⎕TCFF ⋄ ⎕WIN 0 0 1 10 2
```

Type some numbers; then try to type some letters. Notice that you can enter only numbers, a decimal point, and an *E* (for scientific notation) in the field. If the fifth element in the preceding expression is 0, no characters are allowed. Press Esc to exit from ⎕*WIN*.

The sixth column of the right argument to ⎕*WIN* contains the sum of the values of the special features. ⎕*WIN* can perform many interesting and useful tasks if you use and combine these special features. The special features are explained in detail in the "⎕*WIN* Special Features" section later in this chapter.

The seventh and eighth columns contain attributes for the active and inactive fields, respectively. The active field is the field where the cursor sits when ⎕*WIN* is running. All other fields are inactive. Choose the attributes that make it easy for the end user to distinguish the fields from the normal display.

Try this:

Create a variable that defines two fields:

```
      Rwin←2 8⍴0 0 1 6 2 0 45 19 2 20 2 10
1 1 45 119
      Rwin
0  0 1  6 2 0 45  19
2 20 2 10 1 1 45 119
```

The first field is one row and six columns, starting in the upper-left corner. Its field type is 2, so you can only enter numeric data. The second field is two rows and 10 columns, and starts below and to the right of the first field. Its field type is 1, so you can enter any displayable character. Call ⎕*WIN* to see the fields:

```
      ⎕TCFF ⋄ ⎕WIN Rwin
```

If you have a color display, you can see that the two fields have different colors. The first field is the active field and contains the cursor. It is green, because the active field attribute is 45. The second field is the inactive field. It is gray, because the inactive field attribute is 119. Press Tab to move from the first field to the second. Notice how the field display changes, depending on which field is active. Press Esc to exit ⎕*WIN*.

The Left Argument

The left argument to ⎕*WIN* is optional. It controls the entry into ⎕*WIN*. If it is not present, all defaults are used. In that case, ⎕*WIN* begins execution at the top left corner of the first field. It will wait indefinitely in this field until the user presses a key to move out of it. Table 6-2 summarizes the meaning of the elements of the left argument.

Table 6-2. Elements of the Left Argument

Element	Meaning
1	Beginning field; index into right argument
2	Cursor position (row) in current field
3	Cursor position (column) in current field
4	Timeout value for ⎕*WIN* exit
5	Numeric value of keystroke to execute on entry to ⎕*WIN*

The first element specifies the field you begin in after you call ⎕*WIN*. Since this field number is an index into the right argument of ⎕*WIN*, it is sensitive to ⎕*IO*. It is often used after validation to put the cursor back where it was when the exit occurred.

The second and third elements describe the cursor position within the current field. You can make an exit from ⎕*WIN* completely transparent by re-entering ⎕*WIN* with the field number and cursor position as the first three elements of the left argument.

The fourth element is the timeout value in seconds. This number tells ⎕*WIN* how long to wait between keystrokes before exiting. This value defaults to ¯1, which indicates an indefinite wait. You can use this feature to set a limit on the time allowed for a user to answer specific questions. If there is no answer within the specified time, the application can ask if the user needs more time or help, or it can ask for an answer again.

The last element is the event number of a keystroke that is processed when the user enters ⎕*WIN*. There is no default keystroke. If the user is in the field specified by the first element of the left argument and presses the key specified by this event number, ⎕*WIN* can determine what field to move to. With this technique, you do not have to figure out which field the user wants to move to after validating the user's input.

If the validation fails, your application can display an error message. You then re-enter ⎕*WIN* using the last field number and cursor position as the left argument. This technique allows the user to correct the mistake. If the validation succeeds, your application can proceed to the next field. To determine the next field, use the key value of the keystroke that the user pressed to leave the field as the fifth element of the left argument.

Caution: If the user exits a field for validation by filling up the field, the result of ⎕*WIN* contains the last displayable character that the user entered. In this case, you can change the keystroke so that ⎕*WIN* tabs to the next field. The following example illustrates this technique.

Try this:

Create the same fields you created in the last example:

```
        ⎕TCFF ⋄ ⎕WIN Rwin
```

The cursor is in the first field. Press Tab to move between the fields. Press Esc to exit ⎕*WIN*. Now call ⎕*WIN* with the expression:

```
        ⎕TCFF ⋄ 2 ⎕WIN Rwin
```

The cursor is in the second field, because the 2 in the left argument tells ⎕*WIN* to start in the second field. Press Esc. Now try:

```
        ⎕TCFF ⋄ 2 1 4 ⎕WIN Rwin
```

The cursor is now in the second field, second row, fifth column. The 1 and 4 (in origin zero) specify the second row, fifth column. Press Esc. Now try:

```
        ⎕TCFF ⋄ 2 1 4 3 ⎕WIN Rwin
```

After three seconds, ⎕*WIN* exits, because the 3 specifies a timeout of three seconds. Finally, try:

```
        ⎕TCFF ⋄ 2 1 4 ¯1 424 ⎕WIN Rwin
```

The cursor is now in the first field, even though the first element specifies the second field. This is because the fifth element, 424, specifies the Tab key. When you call ⎕*WIN*, this value "presses" Tab so that the cursor jumps back to the first field automatically. (The third element, ¯1, is the default timeout — indefinite wait.

The ⎕*WIN* Result

The result of ⎕*WIN* provides information that you can use to validate input and information you can use in the left argument to ⎕*WIN*. The left argument in subsequent executions of ⎕*WIN* may contain the beginning field, cursor position in that field, and a value for a keystroke that ⎕*WIN* processes. You can also use this information to connect ⎕*WIN* calls with validation code. Although you may call ⎕*WIN* several times, the user sees only a single session. Table 6-3 summarizes the elements in the result.

Table 6-3. Elements of the Result

Element	Meaning
1	Exit code
2	Value of last keystroke before exit
3	Active field on exit
4	Cursor position (row) on exit
5	Cursor position (column) on exit
6-*n*	Indication of changed field (0 or 1); the number of elements matches number of fields

The first element in the result of ⎕*WIN* is a code that describes the type of exit that occurred. These exits generally relate to the use of the special features (described in the next section). Table 6-4 summarizes the exit codes that can occur.

Table 6-4. Exit Codes

Exit Code	Description
0	A key set with ⎕*WKEY* to the action value ¯1 (exit) was pressed.
1	An invalid number was entered. This only occurs if special feature 2 was selected for the field in which the exit occurred.
2	A field was overfilled. This only occurs if special feature 8 was selected.
3	An invalid keystroke was pressed in the current field. You must select special feature 16 for this exit to occur.
4	An exit for validation occurred. This occurs when special feature 32 or 128 is selected.
5	A timeout occurred. If the timeout value supplied in the fourth element of the left argument to ⎕*WIN* has elapsed, ⎕*WIN* exits.
6	A cursor keystroke was pressed to move out of the current field, but no field was found in the direction of the cursor movement. Special feature 1024 must be selected for this exit to occur.
7	An exit bypassing numeric validation occurred. This only occurs if a keystroke valued ¯42 by ⎕*WKEY* is pressed.
8	An exit on arrival occurred. Exit ⎕*WIN* before the current field is entered. Special feature 4096 was selected for the current field.

If a field meets more than one type of exit condition because it has more than one special feature, the highest exit condition is returned. For example, if a field includes the special features for exit on invalid numbers and exit for validation, the exit for validation takes priority. The first element of the result is 4.

The second element is the value of the last keystroke pressed before ⎕*WIN* exited. The third element is the active field when ⎕*WIN* exited. This value is sensitive to the current value of ⎕*IO*. The fourth and fifth elements are the row and column cursor position within the current field when the exit occurred. You can use these values in the left argument of ⎕*WIN* when you call ⎕*WIN* again.

The remaining elements indicate whether each field changed. Cursor movement through a field does not constitute a change, but a displayable-character keystroke does. There are as many elements as fields or rows in the right argument to ⎕*WIN*, so that

```
      (⍴5↓Zwin) = 1⌈1↑¯2↑⍴Rwin
```

is true. If an element is 1, the field was changed. If it is 0, the field was not changed. These bits are reset each time you re-enter ⎕*WIN*. If you want to determine which fields have really changed, even after several exits for validation, you must track the changes yourself.

Try this:

Repeat the previous example. When you press Esc after each example, you see the result of ⎕*WIN*. Notice how the key value, field, cursor position, and exit condition change depending on the right argument to ⎕*WIN* and the keystroke that caused the exit.

Replace the first field's special feature selections (column 6) with 32 to exit for validation and call ⎕*WIN*. Enter some numbers, then press Tab. ⎕*WIN* exits and displays the result:

```
4  424  1  0  5  1  0
```

The first element, 4, means that an exit for validation occurred, as specified by special feature 32. The second element, 424, is the value of the Tab key. The third element, 1, specifies that the exit occurred from field 1. The fourth and fifth elements specify the cursor position at the exit — these may be different, depending on when you pressed Tab. The sixth and seventh fields indicate if the field changed. The 1 indicates that field 1 changed; the 0 indicates that field 2 did not change.

⎕*WIN* Special Features

⎕*WIN* supplies many special feature selections. Table 6-5 summarizes each of the special features. This section discusses each of them in detail. To use more than one special feature in a field, add up the feature numbers and put that sum in column 6 of the right argument to ⎕*WIN*.

If any special features conflict (for example, exit on overfill (8) and discard overfill (64)), the special feature with the **lowest** value has priority. Note that this behavior is different from that of two special features included for a field that cause different return codes to occur on exit. In that case, the **highest** return code when ⎕*WIN* exits is reported.

Table 6-5. ⎕WIN Special Features

No.	Feature	Description
1	Autotab	Default. The cursor moves to the next field automatically when a key is pressed in the last position of the field.
2	Check for Valid Numbers	This feature verifies that the field contains one and only one valid number; if not, ⎕WIN either exits or beeps, depending on whether feature 4 is selected.
4	Remain in Field until Number is Valid	The system beeps and remains in the field until the user enters a valid number. This feature must be used with special feature 2.
8	Exit on Attempt to Overfill Field	This feature causes an exit if a character is typed after the last position in a field is filled or if a character is inserted into a field that forces the last character to "spill" out of the field.
16	Exit on Invalid Keystroke	This feature causes an exit rather than a beep if the entered keystroke is not valid for the field.
32	Exit for Validation	This feature is used for general purpose validation. When the user presses any key that would leave a field with this feature, ⎕WIN terminates and the first element of the result is 4.
64	Discard Overfill	This feature throws away old input when the field is overfilled instead of causing a beep or an exit.

Table 6-5. Continued

No.	Feature	Description
128	Exit when Leaving Modified Field	This feature causes an exit if the field has been changed.
256	No Cursor in Field	The cursor does not display in the field.
512	Clear Field before New Input	This feature clears the field if the user types a displayable character.
1024	Exit if No Field Exists	This feature causes an exit if there is no field in the direction of a geographical keystroke.
2048	Do Not Restore Inactive Screen Attribute	The field retains the active screen attribute if an exit occurs.
4096	Exit on Arrival	This feature causes an exit before a field is entered.

- **Special Feature 1 — Autotab**
  This special feature is the default selection. If a user presses a key when the cursor is in the last position in a field, the cursor automatically moves to the next field. If an exit occurs when the field is filled, the last keystroke the user enters is the second element in the result of ⎕*WIN*. To have your application re-enter ⎕*WIN* after a successful validation, define the left argument to ⎕*WIN* as:

```
        Lwin←Zwin[3 4 5],¯1,
1↑((Zwin[2]≤256)/424),Zwin[2]
```

  424 is the key value for Tab. The cursor moves to the field in the direction indicated by the keystroke.

- **Special Feature 2 — Check for Valid Numbers**
  To verify that there is only one valid number in the current field, ⎕*WIN* uses the same criteria that ⎕*FI* and ⎕*VI* use. If special feature 4 is not also selected, ⎕*WIN* exits and the first element of the result is 1. If you like, you can display a message that prompts the user to enter a valid number and re-enter ⎕*WIN*.

Try this:

Enter the following expression:

```
        ⎕TCFF ◊ ⎕WIN 0 0 1 5 2 2 55 39
```

Enter the invalid expression *E*22 in the field. Now press the Tab key; ⎕*WIN* exits. The first element of the result is 1, which indicates that you entered an invalid number; the second element is 424, the value of the Tab key.

Try combining the autotab special feature with this special feature:

```
        ⎕TCFF ◊ ⎕WIN 0 0 1 5 2 3 55 39
```

Enter *E*2222. When you enter the last 2, ⎕*WIN* exits. The first element of the result is the same, but the second element is 50. In the previous case, the key value corresponded to the Tab key; in this case, it corresponds to the 2 key, key value 50, since that key initiates the autotab feature and causes the exit.

- **Special Feature 4 — Remain in Field until Number Is Valid**
  The feature forces a user to enter valid data in a field. For this feature to work correctly, you must also use feature 2. If a user enters an invalid number, the system beeps and remains in the field. By itself, this feature is not user-friendly. However, if you combine exit on arrival (feature 4096) with this feature, you can display a help message when you enter the field, and the behavior is more helpful.

- **Special Feature 8 — Exit on Attempt to Overfill Field**
  Setting this feature causes an exit if:

  - the user types a character after the last character in a field is entered
  - the user inserts a character that forces characters to "spill" off the edge of the field.

  The first element of the result is 2.

  This feature can be useful you use it with special feature 1024, geographical exit on nonexistent field. For example, the overfill field does not normally cause an exit to occur in the rightmost column of a spreadsheet; adding this feature causes the exit.

Try this:

Enter the expression

```
        ⎕TCFF ⋄ ⎕WIN 0 0 1 5 2 8 55 39
```

Enter the number 2 until ⎕WIN exits. The first element of the result is 2, which indicates that you tried to "overfill" the field.

- **Special Feature 16 — Exit on Any Invalid Keystroke**
  This feature causes an exit if a user enters any keystroke that is not valid for the current field. The first element of the result is 3.

Try this:

Use ⎕*WKEY* to set a new field type, 4, that only allows the letters A, B, C, or D in a field:

```
        4 ⎕WKEY 'ABCD'
```

Now use ⎕*WIN* to create a field. Use 4 as the field type and 16 as the special feature selection:

```
        ⎕TCFF ⋄ ⎕WIN 0 0 1 5 4 16 55 39
```

Enter:

```
        ABCDE
```

As soon as you press E, the invalid character, ⎕*WIN* exits. The result is 3 69 1 0 4 1. The 3 indicates an invalid keystroke; 69 indicates that E was the last keystroke before the exit occurred.

This feature is useful in menu routines where you want to allow the first character of each choice to select that choice. Set the field type of each field at 0 and the special feature at 16. In this case, all of the displayable keystrokes cause an exit, and you can easily check to see if a valid choice was made. Refer to the function *HMENU* in the *WINTUTOR* workspace for an example.

You can also use feature 16 to create "toggle" fields. A toggle field cycles through a list of choices when a user presses a specific key. You can create a routine that recognizes the first letter of each item. The function *ASKYN* in the *WINTUTOR* workspace contains an example.

This special feature also has some use in utility functions. Refer to the functions *WAIT* and *WINKEY* in the *WINTUTOR* workspace.

- **Special Feature 32 — Exit for Validation**
  This feature causes an exit for general-purpose validation. Each time the user presses a keystroke that moves to a different field, ⎕*WIN* terminates and the first element of the result is 4. You could write code that validates that field, then re-enter ⎕*WIN*. If the field is invalid, you can re-enter the same field using

  ```
          Lwin←Zwin[3 4 5] ⋄ →∆WIN
  ```

  where ∆*WIN* is a label on the line where ⎕*WIN* is called. If the validation is successful, you can continue to the next field.

> Try this:
>
> Create a field using this special feature:
>
> ```
>         ⎕TCFF ⋄ ⎕WIN 0 0 1 5 2 32 55 39
> ```
>
> Type some numbers and press Tab to exit ⎕*WIN*. The first element of the result is 4, which indicates special feature 32; the second element is 424, which indicates Tab.

- **Special Feature 64 — Discard Overfill**
  This feature discards input that overfills the field instead of causing a beep or exit. Use this feature if you want the user to be able to type in the last position in a field, but not move to the next field automatically. Features 1 and 8 relate to this feature. Feature 1 moves to the following field; feature 8 exits. If you set both autotab and discard overfill, autotab occurs because the special feature with the lower value has priority.

Try this:

Create two fields, each of which uses special feature 64.

```
      Rwin ← 2 8⍴20 0 1 10 1 64 55 39 22 0
1 10 1 64 55 39
      ⎕TCFF ◊ ⎕WIN Rwin
```

Type some letters in the first field until you overfill the field. Notice how ⎕WIN discards the overfill. Press Tab or Enter to move to the other field. Change the 64 to 65 for the first field and change the 64 to 8 for the second field:

```
      Rwin[1;6]←65
      Rwin[2;6]←8
      ⎕TCFF ◊ ⎕WIN Rwin
```

Type until you overfill the field. The cursor automatically moves to the second field. Because 65 combines special feature 1 with special feature 64, special feature 1 has priority. Now type some letters in the second field. When you overfill the field, ⎕WIN exits. The first element of the result is 2, which indicates that you overfilled the field.

- **Special Feature 128 — Exit when Leaving Modified Field**
  This feature, like feature 32, exits for general validation. The difference is that the exit only occurs if the field has changed. This technique allows you to avoid repeat validation if the field has not changed. However, at the end of the user input session, you may need to check for a value in a field. For example, if a field is empty at the start of a session and a value is required in that field, you must check for a valid value at the end of the session. Cursor movement through a field does not constitute as a change to a field. If you select both 32 and 128, 32 has priority and the system always exits for validation.

- **Special Feature 256 — No Cursor in Field**
  This feature suppresses the display of the cursor and is useful in menu routines and in toggle fields (see feature 16).

- **Special Feature 512 — Clear Field before New Input**
  This feature clears a field immediately if the first keystroke is a displayable character. If the first keystroke is an action key, like a Left or Right cursor, the field is not cleared and is available for editing. This feature is useful if your form contains default information that the user can type over without clearing the remaining characters in that field.

Try this:

```
        ⎕TCFF ◊ ⎕WIN 0 0 1 5 1 512 55 39
```

Enter some values and press Esc. Make sure that the screen has not scrolled, so that your first entry is still in the field. Now, re-enter the *⎕WIN* expression without the *⎕TCFF*. Press a cursor key. Your entry remains in the field. Press Esc and re-enter the expression one more time. Press *A*. The characters in the field disappear.

- **Special Feature 1024 — Exit if No Field Exists**
  Use this feature in conjunction with the geographical cursor movement keys to cause an exit. These keys allow movement to the fields or cursor position directly above, below, left, or right of the current cursor position. If there is no field in the chosen direction, an exit occurs. The first element of the result of *⎕WIN* is 6. (You can set the geographical cursor movement keys with *⎕WKEY* to values ¯34 through ¯37, ¯44 through ¯47, or ¯49 and ¯50. See the section "Interpreting Keystrokes with *⎕WKEY*" later in this chapter for details on using *⎕WKEY*.)

  Spreadsheet applications typically use this feature. For example, if the user moves the cursor off of the current screen, you want an exit to occur so you that can update the screen with other rows or columns of the spreadsheet. You can also use this feature to create a "wrap-around" effect. For example, when you exit from the bottom of the screen, you can determine which field to enter at the top part of the screen.

- **Special Feature 2048 — Do Not Restore Inactive Screen Attribute**
  This special feature causes an active field to retain the active attribute following an exit from ⎕WIN. If your application uses a lengthy validation routine, you can use this feature to keep the active field attribute displayed during validation. You can also use this feature in toggle fields to avoid the flash that occurs when you toggle between selections. You must restore the inactive attribute when you are ready to move to the next field. The expressions

```
      Rwin[Zwin[3];1 2 3 4] ⎕WPUT
Rwin[Zwin[3];8]
```

  or

```
      0 0ρ(Zwin[3],0 0 0) ⎕WIN
Rwin[Zwin[3];]
```

  do this. Note the use of 0 timeout to display attributes.

- **Special Feature 4096 — Exit on Arrival**
  This feature causes an exit before you enter the field. This type of exit differs from others, which exit after you leave a field. The first five elements of the result of ⎕WIN contain:

  - exit code 8
  - the key value for the last keystroke you pressed
  - the field you are about to enter
  - the cursor position 0 0.

  You can use this feature to have your application display a Help message for the field before the user enters information, to calculate and bypass the current field, or to set certain keyboard states before the user enters the field.

  When the application calls ⎕WIN, ⎕WIN ignores the exit on arrival special feature for the initial field in the left argument.

The function *HMENU* is a Lotus-style menu routine that uses this exit on arrival feature. To run the function, type:

```
        anchor HMENU menu
```

Each time you enter a field, ⎕*WIN* exits. If you examine the code immediately following the ⎕*WIN* execution, you can see the check for return code 8 from ⎕*WIN*. If the exit was not exit on arrival, other exits are checked. Otherwise, the Help message displays and you return to the field.

Interpreting Keystrokes with ⎕*WKEY*

⎕*WIN* controls user input in APL★PLUS II/386. ⎕*WKEY* specifies the actions that can be taken when ⎕*WIN* is running and what field types are available for each field. This section explains how to use ⎕*WKEY* to control keyboard actions and define field types.

What is ⎕*WKEY*?

⎕*WKEY* provides two different mechanisms. The first sets keystrokes to perform certain actions in ⎕*WIN*. For example, pressing Home on the default keyboard moves the cursor to the first position in the current field. ⎕*WKEY* allows you to change the behavior of any key to any normal keyboard actions.

The second mechanism associates certain keystrokes with a field type value. For example, you can create a field type that accepts only the digits 0 through 9 for integer-only fields. Other keys cause a beep.

⎕*WKEY* requires a right argument; a left argument is optional. The right argument of ⎕*WKEY* can contain either numeric key values or displayable characters.

If you use only a right argument with ⎕*WKEY*, ⎕*WKEY* returns the actions or field types for each key value in the right argument. If you also use a left argument, ⎕*WKEY* changes the keys in the right argument to the actions or field types you defined in the left argument. The result contains the former actions or field types of each key.

If you set ⎕*WKEY* incorrectly, strange results may occur when you run ⎕*WIN*. If this situation occurs, you may want to reset all of the key values to their defaults and rerun your application.

To reset all of the keys to their defaults, use the function *DEFAULTWKEYS* in the *WINTUTOR* workspace.

Setting Action Keys with ⎕*WKEY*

Pressing "action keys" normally does not cause a display on your monitor screen. Examples of action keys are the Home, PgUp, PgDn, Enter, Esc, and Del keys. ⎕*WKEY* uses negative values to define any accessible key as an action key. This section discusses several of the most important actions. You can run the function *WINKEY* in the *WINTUTOR* workspace to determine the actions of specific keys.

Changing Key Values

By default, the Ctrl-Home and Ctrl-End keys move the cursor to the first and last fields on the screen, (actions ¯4 and ¯5). You might want the Home and End keys to do this as well. To do this, execute the statement:

```
        OLDACTIONS←¯4 ¯5 ⎕WKEY 281 388
```

The next time your application calls ⎕*WIN*, the Home and End keys move to the first and last fields. The values 281 and 388 are the values for the Home and End keys.

In menu routines, you may want the user to use the cursor keys on the numeric pad to move to selections. Normally, the up arrow (↑) stops at the top field and the down arrow (↓) stops at the bottom field. The left (←) and right (→) arrow keys move to the previous and next fields, one space at a time.

If you turn the cursor off in some fields, the user perceives a delay when using these keys. The solution is to change the values of the cursor keys so they become Tab and Backtab keys. To set the up (↑) and left (←) arrow keys as Backtab keys, and the down (↓) and right (→) arrow keys as Tab keys, use the expression:

```
        OLDACTIONS←¯3 ¯3 ¯2 ¯2 ⎕WKEY 393 391
420 389
```

The next time you run ⎕*WIN*, the cursor keys act like Tab keys, and move directly to the next or previous fields.

Sometimes you may want the Del key to erase an entire field instead of just the character at the cursor. Use the expression

```
        OLDACTIONS←¯33 ⎕WKEY 432
```

to accomplish this the next time ⎕*WIN* is run.

Sometimes you may want to disable a key that has some default action. You can disable all of the keys by assigning 0 to them:

```
        OLDACTIONS←0 ⎕WKEY (⍳523)-⎕IO
```

Note: If you use ⎕*WIN* without using ⎕*WKEY* again, you must use Ctrl-Break to exit.

Exiting ⎕*WIN*

The most common action is one that exits from ⎕*WIN*. The "exit from ⎕*WIN*" action has a ¯1 value. Normally, you set keys to ¯1 to allow the user to leave the current screen. Other times, you use ¯1 to exit from ⎕*WIN* to display a Help screen or some submenu or form, and return to the original screen later. This exit is also used in spreadsheet-style routines, so that you can allow PgUp and PgDn to exit, calculate and display the next screen, and re-enter ⎕*WIN*. An exit from ⎕*WIN* is required whenever another available action cannot solve the problem.

Types of Cursor Movement

There are two major types of cursor movement: field movement and geographical movement.

- **Field Movement**
  Field movement is the default. When you press a cursor key, the cursor moves to the closest field in the direction indicated by the key that you pressed. For example, if you place fields in the top left and lower right corners of your screen, the up (↑) and down (↓) arrow keys move to the bottom and top fields, respectively.

Try this:

Create two fields in opposite corners of the screen:

```
      ⎕TCFF ◊ ⎕WIN 2 8⍴0 0 1 10 1 1 55 39
24 60 1 10 1 1 55 39
```

Press the up (↑) and down (↓) arrow keys. Notice that the up arrow (↑) key does not go past the top of the screen and the down arrow (↓) key does not go past the bottom of the screen. Press Esc.

- **Geographical Movement**
  In geographical cursor movement, you can specify the type of movement you want to associate with a certain key. You can set keys to the action values ¯34, ¯35, ¯36, and ¯37 — up, down, left, and right. Keys set with these values move one field at a time. You can use the values ¯44, ¯45, ¯46, and ¯47 to make assigned keys move one cursor position at a time — up, down, left, and right. The first set of keys are useful for menus that spread across a page; the second set is useful for spreadsheet input.

If you assign these values to the keys, the cursor moves to the field or position directly above, below, left, or right of the current field or screen position.

Try this:

Assign the up (↑) and down (↓) arrow keys to actions ¯34 (geographical up) and ¯35 (geographical down):

```
      ¯34 ¯35 ⎕WKEY 393 420
```

Create the same fields you created in the previous example:

```
      ⎕TCFF ⋄ ⎕WIN 2 8⍴0 0 1 10 1 1 55 39
24 60 1 10 1 1 55 39
```

Press the up (↑) and down (↓) arrow keys. Notice that the cursor does not move. There is no field directly above or below each field. Press Esc.

Special feature 1024 causes ⎕*WIN* to exit when there is no available field that can be the target of a geographical move. This makes designing spreadsheet code easier, since an exit occurs whenever the user tries to move off the screen.

Setting Field Types

The system provides three predefined field types: all displayable characters, numeric values, and no displayable characters. The first type, Type 1, is the set of all displayable characters; the second type, Type 2, is the numeric set, which includes the characters "`0123456789.,-¯E`." The third type, Type 0, means that the system does not allow displayable characters. Allowable field type values are always `¯1`, `0`, or powers of 2. You set the field type, either a system field type or a user-defined field type, in Column 5 of the right argument to `⎕WIN`.

Remember: You can associate each character with one or more field types; therefore, you must set all of the field types you want included for each character. For example, the expression

```
        ⎕WKEY 'E'
```

returns the value 3. This result is the sum of the field types assigned to this character. *E* is allowed in Field Type 1, any displayable characters, and Field Type 2, numeric values (for scientific notation). The sum of these field types is 3.

When you create a new field type, include previously defined field types in the new definition. For example, to create a new Field Type 4 for a field that requires a Yes (Y or y) or No (N or n) answer, use the expression:

```
        OLDTYPE←(1+4) ⎕WKEY 'YyNn'
```

(The `(1+4)` is for display only. Use both upper- and lowercase Y and N so that the user does not have to shift to get the correct key. You normally add the field types and use the sum as the left argument.) Notice that you have to include Field Type 1 for the characters in the right argument, so any fields that use Field Type 1 can use those characters. Setting a field to Field Type 4 means that the user can only enter one of the specified characters or press an action key.

You can also create field types to limit numeric values in a field. For example, you create Field Type 8 with:

```
      OLD←(1+2+8) ⎕WKEY '0123456789'
```

Field Type 8 only allows integer values in a field since the right argument does not include a period for the decimal point. Notice that you must include field types 1 and 2. If you do not, those characters are not available in Field Types 1 and 2. You can also create a "positive floating point but no scientific notation" field type by creating Field Type 16 with

```
      OLD←(1+2+16) ⎕WKEY '0123456789.'
```

For both the integer and floating point field types, use

```
      OLD1←(1+2+8+16) ⎕WKEY '0123456789'
```

and

```
      OLD2← (1+2+16) ⎕WKEY '.'
```

The first line creates an integer field type for both 8 and 16. The second line modifies Type 16 by adding the period. Remember: you are assigning the sum of the field type values to each character; the side effect is to create or modify the field types.

The next example shows another way to create field types if you do not know the existing field types:

```
      CHARS←'0123456789.'
      (8+⎕WKEY CHARS)⎕WKEY CHARS
```

This expression "adds" Field Type 8 to all of the keys in *CHARS*.

You can have up to 14 different field types at any one time.

Advanced Techniques and Utilities

This section describes advanced tools. Some topics include examples; others refer to functions in the *VINTUTOR* workspace.

All examples and functions in this section use an index origin of 0. Using `⎕IO←0` in screen design reduces the amount of code and may make your application faster.

Resetting Values Set with ⎕*WKEY*

To avoid confusion, reset values set with ⎕*WKEY* when your application finishes. Follow these steps.

1. Get the current key values.

```
WKEYS←⎕WKEY (⍳523)-⎕IO
```

2. Reset all of the keys to their default values.

```
DEFAULTWKEYS
```

3. Set the field types and action keys for the application.

4. Run the application, then reset the keys to their original values.

```
WKEYS←WKEYS ⎕WKEY (⍳523)-⎕IO
```

Describing Key Values with Variables

A good programming practice is to name the key values as variables, either globally or in an initialization routine.

The expression

```
        →(Zwin[2]=Esc)/⍙Quit
```

is clearer than

```
        →(Zwin[2]=438)/⍙Quit
```

Use the actual names of the keys like Enter, Esc, Home, or PgUp. Call cursor keys Left, Right, Up, and Down. Keystroke combinations can have names like CtrlH or CtrlLeft. Code that uses these conventions is easier to understand, modify, and maintain.

Creating Toggle Fields

A toggle field cycles through a list of choices when a user presses a specific key. Using the "exit on invalid keystroke" feature and a 0 field type (where all keystrokes are invalid) exits a field each time a user presses a key. After you exit, your code can check for the user's keystroke to see if you need to perform some action. Otherwise, re-enter *⎕WIN* at the same field. See the function *ASKYN* in the *VINTUTOR* workspace for an example of this approach.

Follow these steps to create and use a toggle field.

1. Use *⎕WPUT* to display the default characters in the field.
2. Execute *⎕WIN* using special feature 16 with field type 0 for the field.
3. When *⎕WIN* exits, check the keystroke in the second element of the result. If the key value matches the key value of one of the possible selections, prepare to return that value

as a result, in either its character or numeric representation. Using the actual value of a choice is optional. If the choices are more than one character across, use the first character to match.

4. Next, check for some pre-specified "toggle key," such as the Space Bar. If the user pressed this key, display the next option. For example, you could execute the following statements for single character fields:

   ```
           SCR←,Rwin[Zwin[2];⍳4] ⎕WGET 1
           Rwin[Zwin[2];⍳4] ⎕WPUT
   1↑(CHOICES⍳SCR)⌽CHOICES
   ```

   Then branch back to re-enter ⎕*WIN*. For multiple character fields, try this:

   ```
           SCR←,Rwin[Zwin[2];⍳4] ⎕WGET 1
           Rwin[Zwin[2];⍳4] ⎕WPUT
   (¯1⌽CHOICES∧.=SCR)⌿CHOICES
   ```

5. If the user pressed an exit key such as Enter to select or Esc to quit, branch and handle that exit. Otherwise, return to ⎕*WIN* in that same field.

You can enhance these steps by adding feature 2048 to feature 16 for the toggle field. The 2048 causes the active attribute to remain on screen even after the user leaves the field. This avoids the "flash" that can occur when you exit to toggle the field and re-enter as the field switches between active and inactive attributes. See the function *TOGGLE* for a generic toggle routine with other fields. The function *DEMOTOGGLE* demonstrates how to set up a cover function to run the toggle routine.

Waiting for a User to Press Any Key

The function *WAIT* in the *WINTUTOR* workspace provides a way to ask the user to press any key to continue. These points are important to remember.

- Disable all key values to 0.
- Use the 0 field type.
- Use special feature 16, which causes an exit when a user presses any key.
- Use feature 256 to turn off the cursor.
- Finally, reset all of the keys to their original values with *⎕WKEY*.

You can alter this function to suit your own needs. You may want to add a message like "Press any key to continue." You may also want to set one key that displays a Help screen or runs a print routine.

Filling the Screen with Attributes

You can put colors, intensities, or other attributes on different parts of a screen. Normally, you use *⎕WPUT* for each part of the screen you want to color. If you have many parts to color, you have lots of *⎕WPUT* statements; for example:

```
      0 0 3 10 ⎕WPUT 55
      10 0 10 30 ⎕WPUT 39
.
.
.
```

You can use *⎕WIN* to set attributes in multiple rectangular areas on the screen with one line of code. Use a timeout of 0 to have *⎕WIN* display the attributes and exit. The *DISPLAYATTR* function shows this technique. Notice that you do not need field types or special features. The function *RANDISP* uses

DISPLAYATTR to display an unusual pattern on your screen. (Press Esc to stop *RANDISP*.)

Decoding Field Types and Special Feature Values

The field type of a value returned by ⎕*WKEY* and the special feature selection in the sixth column of the right argument to ⎕*WIN* are both sums of values that are always 0 or powers of 2. The function *WINCODE* in the *WINTUTOR* workspace decodes the value associated with a keystroke to determine its field types. It also decodes the special feature selection sum to get the individual special features that describe actions taken for that field.

Determining the Value of a Keystroke

You can use ⎕*WIN* to determine the value of a keystroke. The function *WINKEY* in the *WINTUTOR* workspace is identical to the *WAIT* function, except that it returns the value of the keystroke. This function always returns the correct keystroke to be used with ⎕*WIN*.

Tracking Changed Fields in a ⎕*WIN* Session

You usually use ⎕*WIN* more than once in a given session; for example, to exit for validation, to display a help screen, or to display other information. The change bits after the fifth element of the ⎕*WIN* result show which fields changed in the last execution of ⎕*WIN*. If the field changed earlier in the session, that information is lost.

To avoid end-of-field validation, you usually want to know which fields changed. If you predefine default (or previous) values at the beginning of a session, you can validate the fields that change.

First, define some variable as the overall change bit holder:

```
Changed←(1ρρRwin)ρ0
```

The variable *Changed* contains all 0s since no fields have changed yet. Then run ⎕*WIN* and modify the variable:

```
⍙WIN: Zwin←Lwin ⎕win Rwin
Changed←Changed ∨ 5↓Zwin
```

Insert your validation code, and branch to ⍙*WIN*:

```
→⍙WIN
```

This sequence produces a result that you can use in a routine to retrieve and validate only fields that have been modified in the current session.

What to Save When Moving between Screens

In many applications, you may (1) want to have another menu, screen, or form overlay or "pop-up" on the current form, (2) have the user select or enter some information, and (3) return to the current form. Before you display the "pop-up" screen, you must save the information on the current form; for example:

- keystroke values: *OKEYS*←⎕*WKEY* (⍳523)−⎕*IO*
- function key values changed by the function: *PFKEY*1←⎕*PFKEY* 1 . . . (or use *GETPFKEYS* function)
- the current screen and cursor: *OLDSCR*←⎕*WGET* 3 (or portion of screen being written over) and *OLDCURS*←⎕*CURSOR*

When the "pop-up" form is complete, you must reset the keys, function keys, and the original screen.

Password Entry Using ⎕*WIN*

Use the *PASSWORD* function to allow the user to enter information without displaying it on the screen. *PASSWORD* prevents characters from displaying by using the same foreground and background attribute for ⎕*WIN*. *PASSWORD* uses Field Type 0 and exit on invalid keystrokes to capture the keystrokes. The function checks for an exit key; if there is none, it catenates the character representation of the key value to the result. If the user types the maximum number of characters, or presses Enter or Esc, *PASSWORD* ends and returns the password as the result.

The *WINTUTOR* Workspace

The *WINTUTOR* workspace contains full-screen utility functions and a sample application. The functions whose names begin with *AMS* belong to the sample application; the other functions are independent utilities.

The sample application is an asset management system that uses the following functions.

- ***AMSEXIT***
  This function handles ¯1 and ¯42 exit conditions.
- ***AMSGETSCR***
  This function gets data from the screen and assigns it to variables.
- ***AMSHELP***
  This function calls the Help system for the application.
- ***AMSINIT***
  This function initializes the following items:
  - key variable names
  - field type definitions
  - action keys
  - function key definitions
  - *⎕POKE* values.
- ***AMSINPUT***
  This function is a cover function for the input routine.
- ***AMSIWIN***
  This function sets the arguments to *⎕WIN*.
- ***AMSPUTSCR***
  This function displays the screen with defaults or with values from variables.
- ***AMSRESET***
  This function resets everything initialized with *AMSINIT*.

- *AMSSTORE*
  This function stores screen variables in the global variables.
- *AMSVER*
  This function handles each field's validation.
- *AMSVIN*
  This function runs *⎕WIN* and creates changes vector.

The other functions in the workspace are general utility functions to help you use *⎕WIN*. Most are described in the preceding sections. To see the syntax of the functions, load the *WINTUTOR* workspace and execute *SUMMARY* or *EXPLAIN 'fnname'* where *fnname* is the name of the function you want to examine.

7 Using Advanced APL Language Features

This chapter describes advanced APL language features, particularly nested arrays. The term "nested arrays" refers to a data structure that lets you represent data that are difficult to represent in conventional APL data structures. The chapter is organized as follows.

- "APL★PLUS II/386 Language Evolution" discusses how APL★PLUS II/386 is moving toward compatibility with APL2. It explains how certain language features are changing and describes the mechanism you can use to ensure that your code evolves along with the language.
- "What Are Nested Arrays?" explains the concept of nested arrays and lists the primitive functions you can use to determine the class and equivalence of arrays.
- "Manipulating Nested Arrays" describes the primitive functions you can use to create and modify nested arrays.
- "Using Strand Notation and Strand Notation Assignment" explains what strand notation is and how you use it.
- "Using Existing Functions and Operators" explains how primitive functions and operators work on nested arrays.
- "Using the Each Operator" explains how you use the each operator.

For details on nested arrays functions and operators, refer to the APL Language Reference chapter in the *Reference Manual*.

Note: Other versions of APL use the term "generalized arrays" to refer to nested arrays. Both terms have the same meaning.

This chapter uses the *DISPLAY* function (from the *UTILITY* workspace) to show the scope of a nested array. The system actually uses spaces; for example:

```
      A← 1 2 ∘., 3 4 5
      A
1 3   1 4   1 5
2 3   2 4   2 5
```

The *DISPLAY* function shows the same array as:

```
      DISPLAY A
.→----------------.
↓.→--.  .→--.  .→--.|
||1 3|  |1 4|  |1 5||
|'~--'  '~--'  '~--'|
|.→--.  .→--.  .→--.|
||2 3|  |2 4|  |2 5||
|'~--'  '~--'  '~--'|
'∊----------------'
```

APL★PLUS II/386 Language Evolution

The two most widely used second-generation APL systems are the APL★PLUS System from STSC and the APL2 program product from IBM. Although these two APL systems are similar in how they extend the language, the design and development efforts occurred independently. Because of this independent development effort, the language behavior conflicts in some areas.

The term "conflict" here means a language construct that executes without error but with different effects on both APL★PLUS and APL2. This difference can cause serious errors in code intended to run on both APL★PLUS and APL2.

To resolve this incompatibility, STSC has defined APL★PLUS to eliminate conflicts with the definition of APL2. Beginning with this version of APL★PLUS II/386, STSC is changing the definitions of the features that conflict most seriously with APL2 so that the two systems no longer differ. **This means that you must change existing APL★PLUS II/386 applications if they use features whose definitions are being modified.**

To make the migration of existing APL★PLUS II/386 applications as easy as possible, this version of the system introduces the concept of "evolution levels." This facility allows both old and new definitions of changing features to co-exist for one version of APL★PLUS II/386.

This version of APL★PLUS II/386 implements the first stage of the language evolution by allowing the interpreter to behave at three different evolution levels.

- At Evolution Level 0, the interpreter signals an *EVOLUTION ERROR* when you use a changed feature. Running existing applications at this evolution level allows you to find all of the places where you must change your code.

- At Evolution Level 1, the interpreter uses the old behavior of a changed feature.
- At Evolution Level 2, the interpreter uses the new behavior of a changed feature.

The old behavior for any changing feature lasts for one version only; after that version, only the new behavior will be available.

Table 7-1 shows the language features that have changed and lists the old behavior and new behavior.

Table 7-1. Changed Language Features

Notation	Old Behavior	New Behavior	Alternatives
Monadic ↑	Mix	First	⎕MIX, ⎕FIRST
Monadic ∊	Type	Enlist	⎕TYPE, ⎕ENLIST
Monadic ↓	Split	*SYNTAX ERROR*	⎕SPLIT
Dyadic ⊂	Partitioned Enclose	*NONCE ERROR*	⎕PENCLOSE
`(/)¨`	Replicate Each	*EVOLUTION ERROR*	⎕REPL
`(\)¨`	Expand Each	*EVOLUTION ERROR*	⎕EXPAND
`A B C[2]`	`(A B C)[2]`	*EVOLUTION ERROR*	parentheses
`A B C←X Y Z`	`(A B C)←X Y Z`	*EVOLUTION ERROR*	parentheses

To help you migrate your code, this version of APL★PLUS II/386 contains a mechanism that can diagnose your code and locate changing features. This mechanism includes the `evlevel=` startup parameter and the *)EVLEVEL* system command.

The `evlevel` startup parameter sets the overall state of APL★PLUS II/386:

- `evlevel=0` sets the interpreter to Evolution Level 0
- `evlevel=1` sets the interpreter to Evolution Level 1
- `evlevel=2` sets the interpreter to Evolution Level 2.

The configuration file supplied with APL★PLUS II/386 contains lines with all three versions of the `evlevel` startup parameter. Initially, the switch is set to 1, and changing features use the old behavior. Any code you have that uses the old behavior still works as it always has.

To determine where your code depends on changing features, set the evolution level to 0. When you use this setting and then run your code, any changing features signal an *EVOLUTION ERROR*. You can then update your code to use the new behavior. Once you test your programs and applications for dependencies and change any outdated code, change the configuration setting to Evolution Level 2.

For maximum versatility, you may want to adapt your APL programs so that they work at any evolution level. This is particularly important for utilities that are shared by many applications. To make this possible, several alternative features have been added to the system. These features are system functions that are equivalent to either the old or new definition of a primitive function whose behavior is changing. Code that uses these system functions instead of the primitive runs at any evolution level.

For example, suppose you have an existing function that uses the primitive function mix, which is represented by the monadic up arrow (↑) in previous versions of APL★PLUS II/386. The easiest way to adapt this code is to run it with `evlevel=0`, which causes the code to suspend on an error with each use of monadic up arrow (↑). Once you find where your functions use mix, you can substitute ⎕*MIX* for monadic up arrow (↑) without having to modify your algorithm to use a different feature. With the change, the function runs at any evolution level. Similarly, if you want to use the new primitive function enlist, you can use ⎕*ENLIST* in code that needs to run at any evolution level.

The system command *)EVLEVEL* allows you to change the evolution level during your APL session. Use this command to contrast the new and old behavior of changed features during a session and to search your code interactively for dependencies on old behavior.

Caution: *)EVLEVEL* is intended to help you test code that needs to be changed. You should not use *)EVLEVEL* to dynamically change system behavior in your applications. Such a practice will cause your code to fail in future APL★PLUS II/386 versions.

For example, the definition of monadic epsilon (∊) has changed from the APL★PLUS II/386 type function to the APL2 enlist function. To see the effect of different evolution levels on this feature, use the nested array *V*. (Nested arrays are defined later in this chapter.)

```
      V←3 2⍴'MARY' (2 2⍴4) (⍳5) 0 (2 2⍴'HA') 'JO'
      DISPLAY V
.→--------------------.
↓ .→---.      .→---.  |
| |MARY|      ↓ 4 4|  |
| '----'      | 4 4|  |
|             '~---'  |
| .→---------.        |
| |1 2 3 4 5|  0      |
| '~---------'        |
| .→-.        .→-.    |
| ↓HA|        |JO|    |
| |HA|        '--'    |
| '--'                |
'∊--------------------'
```

At Evolution Level 1, monadic epsilon (ϵ) displays old behavior and acts as the type function:

```
      )EVLEVEL
IS 1
      DISPLAY ϵV
.→--------------------.
↓ .→---.      .→---.  |
| |    |      ↓ 0 0|  |
| '----'      | 0 0|  |
|             '~---'  |
| .→--------.         |
| |0 0 0 0 0|   0     |
| '~--------'         |
| .→-.        .→-.    |
| ↓  |        |  |    |
| |  |        '--'    |
| '--'                |
'ϵ--------------------'
```

Now set the evolution level to 0. Monadic epsilon (ϵ) produces an *EVOLUTION ERROR*:

```
      )EVLEVEL 0
WAS 1
      ϵV
EVOLUTION ERROR
      ϵV
      ^
```

Now set the evolution level to 2, which means that epsilon (ϵ) uses the new behavior. Monadic epsilon (ϵ) acts as the APL2 enlist function. Enlist rearranges an array into a vector, regardless of the array's structure.

```
      )EVLEVEL 2
WAS 0
      ϵV
MARY 4 4 4 4 1 2 3 4 5 0 HAHAJO
      DISPLAY ϵV
.→-----------------------------.
|MARY 4 4 4 4 1 2 3 4 5 0 HAHAJO|
'+-----------------------   -----'
```

You can still access the type function through ⎕*TYPE*:

```
          DISPLAY ⎕TYPE V
.→--------------------.
↓ .→---.      .→---.  |
| |    |      ↓ 0 0|  |
| '----'      | 0 0|  |
|             '~---'  |
| .→----------.       |
| |0 0 0 0 0|   0     |
| '~----------'       |
| .→-.        .→-.    |
| ↓  |        |  |    |
| |  |        '--'    |
| '--'                |
'∊--------------------'
```

Now reset the evolution level to 1.

```
        )EVLEVEL 1
WAS 2
```

The rest of this chapter documents the new behavior of the language features that have changed in Version 4. Where a feature has new behavior, an Evolution Warning notes the old behavior.

The following changes to the language are planned for a future version of APL★PLUS II/386.

- The APL2 partition function will be associated with dyadic enclose (⊂). (At Evolution Level 1, dyadic enclose (⊂) is the partitioned enclose function; at Evolution Level 2, it causes a *NONCE ERROR*.)

- The fill-items used by overtake and expand on matrices and higher-rank arrays will be defined as the typical item of the appropriate row or column, not as the typical item of the entire array. For example,

  ```
        1 0 1\2 2ρ1 2,'AB'
  ```

  will yield

  ```
        2 3ρ1 0 2,'AB'
  ```

  instead of

  ```
        2 3ρ1 0 2,'A',0,'B'
  ```

- The relative precedence of vectors and constants with indexing will be changed to match APL2. (This change will be similar to the change described in the "Using Strand Notation and Strand Notation Assignment" section in this chapter.) This change will affect expressions like:

  ```
        1 2 3 4[3]
  ```

  Use parentheses in your code to make it insensitive to this change; for example,

  ```
        (1 2 3 4)[3]
  or
        1 2 3 (4[3])
  ```

  as appropriate.

What Are Nested Arrays?

The term **nested arrays** describes the concept of **arrays of arrays** or **arrays within arrays**. In conventional APL, each position of an array can contain only a simple scalar. For example, you can create a two-dimensional array (a matrix or table) that has a single number or a single character as each of its items. In a nested array, each position can contain an array of any rank, removing the restriction that each item of an array must be a single number or character. Thus, arrays can be **nested** into other arrays.

The term for the contents of a position of an array is "item." In the following nested array, each item in the array is a three-element vector.

```
      A
 1 2 3  2 4 6  3 6 9  4 8 12
```

The first item contains 1 2 3, the second item contains 2 4 6, the third item contains 3 6 9, and the fourth item contains 4 8 12. The *DISPLAY* function shows each item graphically:

```
      DISPLAY A
.→---------------------------------.
| .→----. .→----. .→----. .→-----. |
| |1 2 3| |2 4 6| |3 6 9| |4 8 12| |
| '~----' '~----' '~----' '~-----' |
'∊---------------------------------'
```

The number of items, not the number of elements in the items, determine the shape of a nested array. Because *A* contains four items, the shape of *A* is 4:

```
      ⍴A
4
```

In the next example, the variable *X* contains a two-item vector. Each item is a two-row, three-column matrix.

```
      X
 1 2 3    7  8  9
 4 5 6   10 11 12

      DISPLAY X
.→-------------------.
|.→----. .→-------.  |
|↓1 2 3| ↓ 7  8  9|  |
||4 5 6| |10 11 12|  |
|'~----' '~-------'  |
'∊-------------------'
```

Because there are two items, the shape of *X* is 2, even though those items are matrices.

```
      ⍴X
2
```

Several functions and an operator help you manipulate nested arrays. Table 7-2 summarizes their behavior for the default evolution level for this version.

Table 7-2. Nested Arrays Functions and Operator, Evolution Level 2

Name	Symbol or Function	Description
Enclose	⊂ (monadic)	Creates a nested scalar.
Partitioned Enclose	⎕*PENCLOSE*	Builds a nonsimple vector from selected portions of an array.
Disclose	⊃ (monadic)	Removes the outermost level of nesting from an array, raising its rank.
Pick	⊃ (dyadic)	Extracts a portion of an array.
Choose	[]	Indexes scattered points of an array.
First	↑, ⎕*FIRST*	Returns the first item of an array.
Enlist	∊, ⎕*ENLIST*	Returns all of the items of an array in a simple vector.
Split	⎕*SPLIT*	Introduces a level of nesting into an array and reduces the array's rank.
Mix	⎕*MIX*	Removes a level of nesting from an array and raises its rank.
Depth	≡ (monadic)	Returns the deepest level of nesting in an array.
Match	≡ (dyadic)	Returns the equivalence of arrays.
Type	⎕*TYPE*	Returns the prototype of an array.
Each (operator)	*f*¨	Applies a function to the items of its argument.

Homogeneous and Heterogeneous Arrays

A simple array is one in which there is no nesting; that is, each position contains an unnested scalar. Simple arrays can be homogeneous or heterogeneous.

- Homogeneous simple arrays contain data of the same type; that is, the data are either all character or all numeric.
- Heterogeneous simple arrays contain mixed data; that is, the same array can contain both character and numeric data.

For example, a typical report contains character data for the row names on a report and numeric data for the rows and columns of the data being reported.

```
      NUMS←2 3 ρ 10 20 30 40 50 60
      NUMS
10 20 30
40 50 60

      NAMES←2 6 ρ 'JAMES JONES '
      NAMES
JAMES
JONES
```

Homogeneous arrays cannot contain both character and numeric data. To generate the complete report in a system that does not support heterogeneous arrays, you must use formatting tools to "tack" the character data and numeric data together, keeping them carefully aligned and synchronized.

```
      '6A1,3LI3' ⎕FMT (NAMES;NUMS)
JAMES 10 20 30
JONES 40 50 60
```

Heterogeneous simple arrays can contain both character and numeric data in any order. In the preceding example, the same matrix could contain both the row names and the numeric data for the rows.

```
      RPT←2 9⍴'JAMES ',10 20 30,'JONES ',40
50 60
      RPT
JAMES  10 20 30
JONES  40 50 60
```

Determining the Class of Arrays

The system function ⎕*TYPE* returns a typical member of the class of arrays to which the argument belongs. You can use this function to determine the datatypes of the items in an array. Arrays are classified by their structure and by the datatypes of their values. A typical member in a class is structured according to the following rules.

- The typical simple numeric scalar is the number 0.
- The typical simple character scalar is a blank.
- The typical array is obtained from the argument by converting each simple scalar in the argument to its typical value.

The argument to ⎕*TYPE* is any array; the result has the structure of that array. Because simple homogeneous arrays must contain the same datatype, the result of ⎕*TYPE* on these arrays contains all 0s or blanks. In the next example, *A* is a simple homogeneous array — a numeric matrix.

```
      A←?2 3⍴10
      A
2 8 5
6 3 1
```

When you use ⎕*TYPE* to determine the class of *A*, the result is all 0s.

```
      ⎕TYPE A
0 0 0
0 0 0
```

If you change *A* to a simple heterogeneous array,

```
      A←'ONLY',4,'YOU'
```

and use ⎕*TYPE* to determine the class, the result consists of blanks for the characters, and a 0 for the number:

```
      ⎕TYPE A
    0
```

To "see" the blanks, use the equal function to generate 1s for the blanks:

```
      A=' '
1 1 1 1 0 1 1 1
```

Evolution Warning: This function was formerly associated with monadic epsilon (∊). At Evolution Level 2, monadic epsilon (∊) is associated with the enlist function. See the "Removing Levels of Nesting" section in this chapter for information on the enlist function.

Determining if Arrays Are Equivalent

The match function (dyadic ≡) compares two arrays and determines if their rank, shape, and values are identical. The arguments can be any rank, shape, and type.

If the arrays are identical, match returns a 1. If not, match returns a 0. The following example compares a scalar 0 to the result of ravel (,) used on 0. Because ravel changes the 0 from a scalar to a one-element vector, the shapes are not identical and match returns a 0:

```
      0≡,0
0
```

The next example compares a character matrix to itself. Because the matrix has not changed, match returns a 1:

```
      A←2 3ρ'CATDOG'
      A≡A
1
```

The next example compares two different character matrices. Although they contain the same characters and have the same rank and shape, the characters are in a different order. Match returns a 0.

```
      B←2 3ρ'DOGCAT'
      A≡B
0
```

Match is useful for verifying that the structure and type of a value are what you expect (for example, an argument to a function). The next example shows this type of usage.

```
      (2 3ρ' ')≡⎕TYPE A
1
```

Match is sensitive to the value of ⎕CT.

Manipulating Nested Arrays

This section explains how you create nested arrays, add and remove levels of nesting, and select individual items from a nested array.

Creating a Nested Array

The enclose (monadic ⊂) function turns any array, except a simple scalar, into a nested array. If the array is already nested, enclose adds a level of nesting to it. A simple scalar is returned unchanged.

The right argument is the array you want to enclose; the result is the same array with an added layer of nesting.

```
      B←⊂1 2 3
      DISPLAY B
.-------.
|.→----.|
||1 2 3||
|'~----'|
'∊------'

      ⍴⍴B
0

      ⊂⊂1 2
 1 2

      DISPLAY ⊂⊂1 2
.-------.
|.-----.|
||.→--.||
|||1 2|||
||'~--'||
|'∊----'|
'∊------'
```

```
      ⊂4
4
      C←(⊂'WHAT'),(⊂'HATH'),(⊂'GOD'),(⊂'WROUGHT')
      ⍴C
4
      DISPLAY C
.→----------------------------------.
| .→---. .→---. .→--. .→------.     |
| |WHAT| |HATH| |GOD| |WROUGHT|     |
| '----' '----' '---' '-------'     |
'∊----------------------------------'
```

If you want to create a nested array from certain portions of an array, use the system function ⎕PENCLOSE (partitioned enclose).

The right argument is the array from which you want to select items to enclose. It can have any rank. The left argument is a Boolean vector used to partition the right argument into sections. It must be the same length as the selected coordinate of the right argument. Each section begins with a 1 and ends before the next 1.

In the following example, the first two elements of the array are not selected, 3 4 become the first item, 5 becomes the second item, and 6 7 8 become the third item.

```
      A←0 0 1 0 1 1 0 0 ⎕PENCLOSE ⍳8

      DISPLAY A
.→-------------------.
| .→--. .→. .→----.  |
| |3 4| |5| |6 7 8|  |
| '~--' '~' '~----'  |
'∊-------------------'

      ⍴A
3
```

If the left argument is a scalar, ⎕*PENCLOSE* reshapes it to match the selected coordinate of the right argument.

```
      DISPLAY 1 ⎕PENCLOSE 1 2 3
.→-----------.
|.→. .→. .→.|
||1| |2| |3||
|'~' '~' '~'|
'∊-----------'
```

You can also specify the coordinate of an array. To do so, enclose the coordinate in brackets ahead of the array. In the next example, the new array is constructed from a three-row, four-column matrix by enclosing the first two rows as one item and the last row as another item.

```
      DISPLAY 1 0 1 ⎕PENCLOSE[1] 3 4⍴⍳12
.→-----------------------.
|.→------. .→-----------.|
|↓1 2 3 4| ↓9 10 11 12||
||5 6 7 8| '~-----------'|
|'~------'               |
'∊-----------------------'
```

Evolution Warning: At Evolution Level 1, the partitioned enclose function is associated with dyadic enclose (⊂). At Evolution Level 2, dyadic enclose (⊂) returns a *NONCE ERROR*. A future version of APL★PLUS II/386 will replace this error with the partition function, as defined by APL2.

Adding Additional Levels of Nesting

The system function ⎕*SPLIT* encloses a dimension of an array. ⎕*SPLIT* returns a simple scalar unchanged; it encloses a nested scalar once more. When you enclose an array with ⎕*SPLIT*, the rank of the result is one less than the rank of the original array. Each item of the result is a vector with the length of the enclosed dimension.

The next example encloses each row of a matrix and returns a vector of vectors.

```
      DISPLAY ⎕SPLIT 3 4⍴⍳12
.→----------------------------------.
| .→------. .→------. .→----------. |
| |1 2 3 4| |5 6 7 8| |9 10 11 12| |
| '~------' '~------' '~----------' |
'∊----------------------------------'
```

You can specify the dimension along which to enclose the array, as shown in the next example. Here, the matrix is enclosed along the columns. The dimension is specified in brackets right after the function.

```
      DISPLAY ⎕SPLIT[1] 3 4⍴⍳12
.→--------------------------------------.
| .→----. .→-----. .→-----. .→-----. |
| |1 5 9| |2 6 10| |3 7 11| |4 8 12| |
| '~----' '~-----' '~-----' '~-----' |
'∊--------------------------------------'
```

You can also use ⎕*SPLIT* to enclose multi-dimensional arrays along a specified coordinate. The next example turns a rank 3 matrix into a vector of rank 2 matrices.

```
      A←⎕SPLIT 2 3 4⍴⍳24
      DISPLAY A
.→-----------------------------------------------.
↓ .→------.      .→------.      .→----------.    |
| |1 2 3 4|      |5 6 7 8|      |9 10 11 12|    |
| '~------'      '~------'      '~----------'    |
| .→----------. .→----------. .→----------.      |
| |13 14 15 16| |17 18 19 20| |21 22 23 24|      |
| '~----------' '~----------' '~----------'      |
'∊-----------------------------------------------'
      ⍴A
2 3
      ⍴⍴A
2
```

Evolution Warning: At Evolution Level 1, split is associated with monadic down arrow (↓). At Evolution Level 2, monadic down arrow (↓) returns a *SYNTAX ERROR*.

Removing Levels of Nesting

APL★PLUS II/386 provides several ways for you to remove levels of nesting from a nested array. This section discusses disclose, first, mix, and enlist.

Disclose

The disclose function (monadic ⊃) undoes the work of the enclose function. It removes the outermost layer of nesting from a nested scalar and returns the item.

```
      B←⊂1 2 3

      DISPLAY B
.-------.
| .→----. |
| |1 2 3| |
| '~----' |
'∊-------'
```

```
      DISPLAY ⊃B
.→----.
|1 2 3|
'~----'

      ρ⊃B
3

      ⊃⊂'ABC'
ABC
      DISPLAY ⊃⊂'ABC'
.→--.
|ABC|
'---'

      ⊃⊂5
5

      ⊃5
5

      A←(⊂1 2 3),(⊂2 4 6),(⊂3 6 9),(⊂4 8 12)
      DISPLAY A
.→---------------------------------------.
| .→----.  .→----.  .→----.  .→-----.    |
| |1 2 3|  |2 4 6|  |3 6 9|  |4 8 12|    |
| '~----'  '~----'  '~----'  '~-----'    |
'∊---------------------------------------'

      DISPLAY A[2]
.-------.
| .→----. |
| |2 4 6| |
| '~----' |
'∊-------'

      DISPLAY ⊃A[2]
.→----.
|2 4 6|
'~----'
```

If each item in a nested vector has the same shape, you can use disclose to turn the nested vector into a matrix. Each item in the nested vector becomes a row in the result matrix; for example:

```
      DISPLAY A
.→-------------------------------.
|.→----. .→----. .→----. .→-----.|
||1 2 3| |2 4 6| |3 6 9| |4 8 12||
|'~----' '~----' '~----' '~-----'|
'∊-------------------------------'

      ⊃A
 1 2  3
 2 4  6
 3 6  9
 4 8 12

      ρA
[illegible]
      ρ¨A
3 3 3 3
      ρ⊃A
4 3
```

Evolution Warning: At Evolution Level 1, disclose does not turn a nested vector into a matrix. When you applied disclose to an array that has more than one item, it returned only the first item. At Evolution Level 2, this behavior is a distinct primitive function call first, and is associated with monadic up arrow (↑). See the "First" section for more information.

First

The first function returns the first item of an array that has more than one item. Monadic up arrow (↑) and the system function ⎕*FIRST* both act as the first function.

```
      ↑2 3 4
2
      ⎕FIRST A
1 2 3
```

Evolution Warning: At Evolution Level 1, the first function is associated with disclose (⊃) applied to an array with more than one item. At Evolution Level 1, monadic up arrow (↑) is associated with the mix function.

Mix

The system function ⎕*MIX* undoes the work of ⎕*SPLIT*. It assembles the items of its argument and inserts dimensions between the dimensions that you specify. If you do not specify the dimensions, the additional coordinates are catenated to the right end of the shape vector, as shown in the next example.

```
      DISPLAY ⎕MIX (1 2 3 4)(5 6 7 8)(9 10
11 12)
.→----------.
↓1  2  3  4|
|5  6  7  8|
|9 10 11 12|
'~----------'
```

To specify the dimensions, use a fractional value enclosed in brackets directly after the function. For example, to insert the coordinate before dimension 1, you use the value .5.

```
      DISPLAY ⎕MIX [0.5](1 2 3 4)(5 6 7
8)(9 10 11 12)
.→-----.
↓1 5  9|
|2 6 10|
|3 7 11|
|4 8 12|
'~-----'
```

The preceding example shows how mix turned a vector of vectors into a matrix.

Evolution Warning: At Evolution Level 1, the mix function is associated with monadic up arrow (↑). At Evolution Level 2, monadic up arrow (↑) acts as the first function.

Enlist

The enlist function turns any array into a simple vector. At Evolution Level 2, this function is available as monadic epsilon (*∊*); the system function *⎕ENLIST* is available at any evolution level. Enlist does not preserve rank, shape, or depth of the array.

```
      A←'MARY' (2 3⍴⍳6) 0 'JOE'
      DISPLAY A
.→------------------------.
| .→---. .→-----.   .→--. |
| |MARY| ↓ 1 2 3| 0 |JOE| |
| '----' | 4 5 6|   '---' |
|        '~-----'         |
'∊------------------------'
```

Enlist turns this four-element, nested vector into a simple heterogeneous vector:

```
     DISPLAY ⎕ENLIST A
.→---------------------.
|MARY 1 2 3 4 5 6 0 JOE|
'+---------------------'

      ⍴A
4
      C←∊A
      ⍴C
14
```

Evolution Warning: At Evolution Level 1, monadic epsilon (*∊*) is associated with the type function, and only *⎕ENLIST* acts as the enlist function. At Evolution Level 2, monadic epsilon (*∊*) acts as enlist; *⎕TYPE* acts as the type function.

Determining Levels of Nesting

The depth function (monadic −) measures the levels of nesting in an array. Simple scalars have a depth of 0; simple non-scalars have a depth of 1. Nested arrays have a depth greater than 1.

The argument can have any rank, shape, or type. The result is a simple numeric scalar.

```
      ≡3.3
0
      ≡1 2 3
1
      ≡⊂1 2 3
2
      ≡2 3⍴'CATDOG'
1
      ≡⊂⊂⊂⊂⊂1 2
6
```

Selecting Items from a Nested Array

The pick function (dyadic ⊃) extracts a portion of an array. Each successive item of the left argument selects a more deeply nested item of the right argument.

```
      DISPLAY A
.→----------------------------------------.
| .→----.  .→----.  .→----.  .→-----.     |
| |1 2 3|  |2 4 6|  |3 6 9|  |4 8 12|     |
| '~----'  '~----'  '~-↑--'  '~-----'     |
'∊---------------------↑------------------'
      3⊃A
3 6 9
      3 2⊃A
6     ↑
```

In the preceding example, the 3 picks the third item of *A*, which is the vector 3 6 9, and the 2 picks the second element of that vector, the 6.

When picking from an array that is not a vector, the corresponding item of the left argument must have a shape equal to the rank of the array. For instance, to select an item from a matrix, the corresponding item in the left argument must be a two-item vector.

```
      DISPLAY X
.→-----------------.
|.→----. .→--------.|
|↓1 2 3| | 7  8  9||
||4 5 6| |10 11 12||
|'~----' '~--------'|
'∊-----------------'

      DISPLAY 1⊃X
.→----.
↓1 2 3|
|4 5 6|
'~----'

      DISPLAY (1,⊂2 3)⊃X
 6
```

Indexing Points in an Array

The choose function extends traditional APL indexing so that you can pull several scattered points out of an array in one operation. This form of indexing uses a nested array to specify the points in the array you are indexing.

In regular indexing, you use simple scalars as indices. When indexing into a matrix, you separate the row and column indices with a semicolon. In choose indexing, however, the indices within the brackets are not simple; rather, they are items in a nested array. Each item in the array is a vector that forms one index.

The array you index can have any rank. The argument within the brackets is a nested array, each item of which is a vector. The length of each item must match the rank of the array being indexed.

In the following example, *A* is a three-row, four-column matrix. Because *A* is a matrix, it has a rank of 2.

```
      A←3 4ρι12
      A
 1  2  3  4
 5  6  7  8
 9 10 11 12
```

Note: The next examples use a type of notation know as strand notation. For a detailed explanation of strand notation, see the "Using Strand Notation and Strand Notation Assignment" section later in this chapter.

In traditional indexing, you use the following expression to select the 10, 4, and 7 from the matrix *A*:

```
      A[3;2],A[1;4],A[2;3]
10 4 7
```

This expression indexes *A* three times and catenates the results. The next expression uses choose to select the 10, 4, and 7 from the matrix *A*:

```
      A[(3 2)(1 4)(2 3)]
10 4 7
```

Notice that each item in the nested array used as the index has a length of 2, the same as the rank of *A*.

You can also use choose to assign new values to scattered points in an array. The next example replaces the major diagonal in the matrix *A*:

```
      A[(1 1)(2 2)(3 3)]←0
      A
 0  2  3  4
 5  0  7  8
 9 10  0 12
```

In traditional indexing, you receive an *INDEX ERROR* when you try to index a scalar. Choose allows you to index scalars using enclosed empty numeric vectors; for example:

```
      B←5
      B[⊂⍳0]
5
      CHAR←'Z'
      CHAR[⊂⍳0]
Z
```

The result of using choose to index a scalar is the value of the scalar.

Replacing Data in Arrays

Selective specification is a form of assignment that lets you use an expression on the left side of an assignment arrow. This expression specifies the portion of the array to be replaced by the data on the right side of the assignment arrow.

In the following example, the selection expression (3⊃*A*) replaces the 0 in the third item of *A* with the character matrix *HAHA*.

```
      A←'MARY' (2 3⍴⍳6) 0 'JOE'
      DISPLAY A
.→--------------------------.
| .→---. .→-----.   .→--.   |
| |MARY| ↓ 1 2 3| 0 |JOE|   |
| '----' | 4 5 6|   '---'   |
|        '~-----'           |
'∊--------------------------'

      (3⊃A)←2 2⍴'HA'
      DISPLAY A
.→--------------------------------.
| .→---. .→-----. .→-. .→--.      |
| |MARY| ↓ 1 2 3| ↓HA| |JOE|      |
| '----' | 4 5 6| |HA| '---'      |
|        '~-----' '--'            |
'∊--------------------------------'
```

The (3⊃A) selection expression uses disclose to return the 0 from the array *A*. The expression 2 2⍴'HAHA' contains the replacement for the 0.

The variable name at the right end of all selection expressions is the variable whose contents you want to replace. The rest of the selection expression evaluates to the part of the variable to be replaced. The expression on the right side of the assignment arrow specifies the replacement.

The preceding example used selective specification to replace one complete array in the variable *A*. You can also replace subsets of arrays. If the replacement data are a subset, the shape of the subset must conform to the shape of the replacement array. The arrays conform if their shapes match or if the replacement array is a scalar.

If the replacement array has only one element, that element is assigned to each element of the selection array; otherwise, corresponding elements of the replacement array are assigned to the locations of the selection array.

The next example replaces the first element of the first item, the letter *M*, with the single letter *W*.

```
      (1↑1⊃A)←'W'
      DISPLAY A
.→---------------------------.
| .→---. .→-----. .→-. .→--. |
| |WARY| ↓ 1 2 3| ↓HA| |JOE| |
| '----' | 4 5 6| |HA| '---' |
|        '~-----' '--'       |
'∊---------------------------'
```

Even though the *M* is a one-element vector, the *W* is a scalar and therefore conforms. However, the next example results in an error:

```
      (1↑4⊃A)←'SL'
LENGTH ERROR
      (1↑4⊃A)←'SL'
      ^        ^
```

The selection expression evaluates to the one-element vector *J*, but the *SL* is a two-element vector. The error occurs because the shapes do not conform. If you enclose the replacement array, the array become a scalar. The new replacement replaces the *J* with the nested vector *SL*:

```
      (1↑4⊃A)←⊂'SL'
      DISPLAY A
.→---------------------------------------.
| .→---.  .→-----.  .→-.  .→---------.   |
| |WARY|  ↓ 1 2 3|  ↓HA|  | .→-.     |   |
| '----'  | 4 5 6|  |HA|  | |SL| O E |   |
|         '~-----'  '--'  | '--' - - |   |
|                         'ϵ---------'   |
'ϵ---------------------------------------'
```

Some functions, such as take and expand, may introduce "fill" elements. When you use these functions in a selection expression, the fill elements specify locations that absorb elements of the replacement data. In the next example, the 0s absorb the second and fourth elements of the replacement data.

```
      X←'ABC'
      (1 0 1 0 1\X)←'VWXYZ'
      X
VXZ
```

You can use the following primitive functions in selection expressions:

- drop (↓)
- enlist (ϵ, ⎕ENLIST)
- expand (\, ⎕EXPAND)
- monadic and dyadic transpose (⍉)
- ravel (,)
- replicate (/, ⎕REPL)
- reshape (⍴)
- reverse first dimension (⊖) and reverse last dimension (⌽)
- rotate first dimension (⊖) and rotate last dimension (⌽)
- take (↑).

Using Strand Notation and Strand Notation Assignment

This section explains how you use strand notation to enter vectors (either simple or nested) and how you use strand notation assignment to assign more than one variable in a single operation.

Note: Other APL systems refer to strand notation as vector notation.

What is Strand Notation?

Strand notation is a means of entering vectors, either simple or nested. Three kinds of constructs appear in strand notation:

- constant numeric values such as `12` or `1 2 3`
- constant character values such as `'A'` or `'HIERONYMUS BOSCH'`
- expressions such as `(PICKLE×JUICE)`.

When two or more of the constructs are adjacent, each is interpreted as an item.

Constructs that evaluate to simple scalars remain simple.

Strand notation is an extension of the familiar notation used to enter a constant numeric vector. A position can now consist of a number or character (as before), any rank or shape array, or an expression. An expression may need to be enclosed in parentheses to limit the scope of the functions within it.

Note that stranding occurs only when two or more values are adjacent.

All of the following examples (excluding the initial assignment) return three-item vectors.

```
      A←1 ◊ B←2 ◊ C←3 ◊ D←1 2 3
      A B C
1 2 3

      DISPLAY A B D
.→----------.
|     .→----.|
|1 2  |1 2 3||
|     '~----'|
'∊-----------'

      A B C×2
2 4 6

      DISPLAY A B D+10
.→-------------------.
|       .→---------. |
|11 12  |11 12 13| |
|       '~---------' |
'∊-------------------'

      DISPLAY A B (D+10)
.→--------------.
|    .→-------. |
|1 2 |11 12 13||
|    '~-------'|
'∊--------------'

      DISPLAY (1 9 4 1) 4 'YOU'
.→-----------------.
|.→------.    .→--.|
||1 9 4 1| 4  |YOU||
|'~------'    '---'|
'∊-----------------'

      DISPLAY A 'SNARK' 3.14
.→-------------.
|  .→----.     |
|1 |SNARK| 3.14|
|  '~----'     |
'∊-------------'
```

```
          DISPLAY (2 3) 4 5
.→---------.
| .→--.    |
| |2 3|  4 5|
| '~--'    |
'∊---------'

          DISPLAY 5 '=' 'V'
.→---.
|5 =V|
'+---'

          DISPLAY 5 '=V'
.→-----.
|   .→-.|
|5  |=V||
|   '--'|
'∊------'
```

If you find strand notation difficult to understand in certain constructs, you can use parentheses to make the expression clearer; for example:

```
          DISPLAY (A B D)+10
.→----------------.
|       .→--------.|
|11 12  |11 12 13||
|       '~--------'|
'∊----------------'
```

Using Strand Notation Assignment

Strand notation assignment allows you to assign more than one variable in one operation; for example:

```
          R←1
          (C D E) ← R
          C D E
1 1 1
```

Each variable to the left of the assignment arrow receives the corresponding item of the vector to the right. Up to 255 names can appear to the left of the assignment arrow. A scalar or one-item vector right argument is changed into a vector with one item for each variable name on the left.

The right argument is a vector with as many items as there are names to the left of the assignment arrow. Some examples follow.

```
      (A B C)←1 2 3
      A ⋄ B ⋄ C
1
2
3

      (A B C) ← 4
      A ⋄ B ⋄ C
4
4
4

      (A B C) ← ⊂1 2 3
      A ⋄ B ⋄ C
1 2 3
1 2 3
1 2 3

      DISPLAY A ⋄ DISPLAY B ⋄ DISPLAY C
.→------.
|.→----.|
||1 2 3||
|'~----'|
∊-------'
.→------.
|.→----.|
||1 2 3||
|'~----'|
∊-------'
.→------.
|.→----.|
||1 2 3||
|'~----'|
∊-------'
```

In the next example, the right argument of *FNCHANGE* is broken into its two constituent parts; that is, the string to change from and the string to change to.

```
     ∇ FNS FNCHANGE REPL;...
[1]

[2]   L1:(OLD NEW)←REPL ⋄ ..
```

Evolution Warning: Evolution Level 1 does not require parentheses around the expressions on the left side of the assignment. Evolution Level 2 requires parentheses.

Using Functions and Operators with Nested Arrays

Scalar functions apply through all levels of nesting.

```
      DISPLAY A
.→-----------------------------------.
| .→----. .→----. .→----. .→-----.   |
| |1 2 3| |2 4 6| |3 6 9| |4 8 12|   |
| '~----' '~----' '~----' '~-----'   |
'∊-----------------------------------'

      DISPLAY A+A
.→---------------------------------------.
| .→----. .→-----. .→-------. .→-------. |
| |2 4 6| |4 8 12| |6 12 18| |8 16 24| |
| '~----' '~-----' '~-------' '~-------' |
'∊---------------------------------------'

      DISPLAY A÷1 2 3 4
.→-----------------------------------.
| .→----. .→----. .→----. .→----.    |
| |1 2 3| |1 2 3| |1 2 3| |1 2 3|    |
| '~----' '~----' '~----' '~----'    |
'∊-----------------------------------'

      DISPLAY A-1
.→-----------------------------------.
| .→----. .→----. .→----. .→-----.   |
| |0 1 2| |1 3 5| |2 5 8| |3 7 11|   |
| '~----' '~----' '~----' '~-----'   |
'∊-----------------------------------'

      DISPLAY 1 2 3 4×⊂1 2 3
.→-----------------------------------.
| .→----. .→----. .→----. .→-----.   |
| |1 2 3| |2 4 6| |3 6 9| |4 8 12|   |
| '~----' '~----' '~----' '~-----'   |
'∊-----------------------------------'
```

In the last two examples, the right argument to minus (-) and times (×) is a scalar. Following the rules of scalar extension, the right argument is replicated to match the shape of the left argument before the function is performed.

Structural functions apply to the outermost level of a nested array.

```
      DISPLAY 2 2⍴A
.→----------------.
↓ .→----.  .→----. |
| |1 2 3|  |2 4 6| |
| '~----'  '~----' |
| .→----.  .→-----.|
| |3 6 9|  |4 8 12||
| '~----'  '~-----'|
'∊-----------------'

      DISPLAY ⌽A
.→-----------------------------------------.
| .→-----.  .→----.  .→----.  .→----.  |
| |4 8 12|  |3 6 9|  |2 4 6|  |1 2 3| |
| '~-----'  '~----'  '~----'  '~----' |
'∊-----------------------------------------'
```

Derived functions can also be applied to nested arrays. In the following example, the addition function has been applied to the items of *A* in the pattern specified by the reduction and scan operators.

```
      DISPLAY +/A
.→----------.
| .→--------.|
| |10 20 30||
| '~--------'|
'∊----------'

      DISPLAY +\A
.→--------------------------------------------------.
| .→----.  .→----.  .→-------.  .→--------. |
| |1 2 3|  |3 6 9|  |6 12 18|  |10 20 30||
| '~----'  '~----'  '~-------'  '~--------'|
|∊--------------------------------------------------'
```

Operators can call any function; in particular, user-defined functions, primitive functions, derived functions, and system functions.

When operators call user-defined functions, powerful yet easily understood array-oriented control structures provide for function calls. You can also use this feature to explore the behavior of an operator, as in the following example.

```
      ∇ Z←L MINUS R
[1]     Z←L-R
[2]     'I2,<->,I2,<=>,I2' ⎕FMT 1 3⍴L R Z
      ∇
        5 MINUS 3
 5- 3= 2
2
        -/⍳5
3
        MINUS/⍳5
 4- 5=¯1
 3-¯1= 4
 2- 4=¯2
 1-¯2= 3
3
```

Operators are not restricted to calling scalar functions.

```
        DISPLAY ,/'MARES' 'EAT' 'OATS'
.---------------.
| .→-----------. |
| |MARESEATOATS| |
| '------------' |
'∊---------------'
```

The next example shows one of the most useful idioms of nested arrays. Disclose of the catenate-reduction (⊃,/) takes a nested vector of simple vectors and returns a simple vector with nesting removed. (You can also use enlist to achieve this effect.)

```
        DISPLAY ⊃,/'MARES' 'EAT' 'OATS'
.→-----------.
|MARESEATOATS|
'------------'
```

```
      DISPLAY 1 2 3 ∘., 4 5
.→-----------.
↓.→--.  .→--.|
||1 4|  |1 5||
|'~--'  '~--'|
|.→--.  .→--.|
||2 4|  |2 5||
|'~--'  '~--'|
|.→--.  .→--.|
||3 4|  |3 5||
|'~--'  '~--'|
'∊-----------'
```

Operators have restrictions about the valence (the number of arguments) of the functions that are their arguments; for example, reduction requires that its function argument be **dyadic.**

Using the Each Operator

The each operator (¨) applies a function to the items of its argument or between the items of its arguments to produce the items of its result. Each provides the power of scalar extension to any function; however, such scalar extension applies only to the outermost level of a nested array.

```
      DISPLAY 1 2 3 ⍴¨ 4 5 6
.→----------------.
|.→. .→--. .→----.|
||4| |5 5| |6 6 6||
|'~' '~--' '~----'|
'∊----------------'

      DISPLAY 1 2 3 ,¨ 4 5 6
.→----------------.
|.→--. .→--. .→--.|
||1 4| |2 5| |3 6||
|'~--' '~--' '~--'|
'∊----------------'

      R←(⊂2 3 5),⊂7 11 13
      DISPLAY R
.→-----------------.
|.→----. .→-------.|
||2 3 5| |7 11 13||
|'~----' '~-------'|
'∊-----------------'

      DISPLAY ⌽R
.→-----------------.
|.→-------. .→----.|
||7 11 13| |2 3 5||
|'~-------' '~----'|
'∊-----------------'

      DISPLAY ⌽¨ R
.→-----------------.
|.→----. .→-------.|
||5 3 2| |13 11 7||
|'~----' '~-------'|
'∊-----------------'
```

```
      DISPLAY ⌽ ⌽¨ R
.→-----------------.
| .→------. .→----. |
| |13 11 7| |5 3 2| |
| '~------' '~----' |
'∊------------------'
```

The each operator can accept as its argument a derived function such as plus-reduction. In fact, any operator can accept any derived function.

```
      +/¨ R
10 31
```

The next example builds a five-item vector, each of whose items is a two-item vector. Each two-item vector is used as an argument to the ⎕FREAD function. The result is a five-item vector, each of whose items is a component read from the file.

```
      FILE←⎕FREAD¨ ⎕← 2 ,¨ ⍳5
 2 1  2 2  2 3  2 4  2 5
```

In the next example, *TIENO* is catenated, item-by-item, to the items of ⍳10. The result is a ten-item vector, each item of which is, itself, a two-item vector. Next, the function ⎕FREAD is applied to each two-item vector. The final result is a 10-item vector that contains the first 10 components of the file tied to *TIENO*.

```
      C←⎕FREAD¨TIENO,¨⍳10
```

8 Using Graphics Capabilities

This chapter explains how you use two different options to generate graphics in APL★PLUS II/386: ⎕G-style graphics and VDI graphics. Each approach has different features and benefits; you must decide which one suits your application.

The ⎕G-style graphics are enhanced versions of the graphics available in ROM BASIC. They take advantage of APL's array-handling capabilities.

The VDI graphics provide a higher degree of support for a broader range of output devices.

The following sections describe the two approaches in more detail. They provide the information you need to decide which form of graphics you should use.

Using ⎕*G*-Style Graphics

APL★PLUS II/386 supports graphics with a number of system functions. These functions provide direct access to the graphics capabilities of your personal computer from within the APL environment. This section presents background and general considerations for using graphics. It discusses hardware considerations, the display environment, and the graphics system functions. The actual system functions (all begin with ⎕*G*) are described in detail in the *Reference Manual.*

What Are ⎕*G*-Style Graphics?

The ⎕*G*-style graphics system functions are based on the graphics functions in ROM BASIC. You can use them to create lines, curves, boxes, and circles. You can control the colors, shading patterns, and interaction of the different objects placed on the graphics easel.

Because they are system functions, you need not include any primitive graphics support functions in an application workspace. You can generate a single function to carry out a graphics operation that can be copied from one workspace to another.

Hardware Considerations

The graphics system functions are implemented for many of the popular graphics adapter boards currently available for personal computers. You use the system function ⎕*GINIT* to configure the APL graphics software to most common graphics adapters.

Most display devices used in personal computers (such as CGA, EGA, VGA, and Hercules) operate in either graphics mode or text mode. Text mode is the default mode for hardware and is assumed by APL★PLUS II/386. You cannot produce graphics in text mode. In graphics mode, text output has no effect or produces garbage on the graphics image. Use ⎕*GWRITE* to produce alphanumeric characters in graphics mode. Ordinary APL output (including default display, quad (⎕), quote-quad (⍞), ⎕*WPUT*, and ⎕*ARBIN*) requires text mode.

Because you cannot mix text and graphics mode, you should produce all graphics under program control. Follow these steps:

1. Initialize graphics with ⎕*GINIT*.

2. Execute the graphics functions.

3. Return to immediate execution mode and text mode with 0 ⎕*GINIT* 0. You can also return to text mode and immediate execution mode with Ctrl-Esc while testing graphics display programs.

If you use a single display monitor, write text to the screen during graphics display with ⎕*GWRITE*. Any non-graphics output or input that displays on the screen will execute correctly, but will look strange on the screen. In particular, interactive functions should use ⎕*INKEY* or ⎕*ARBIN* from ports 11, 16, 17, 18, or 19 (instead of quad (⎕) or quote-quad (⍞)) to accept input from the keyboard.

The restriction on mixing text output and graphics on the same screen is relaxed for two common graphics modes: IBM color card (or compatibles) in two-color 640-by-200 resolution, and the Hercules monochrome graphics card in two-color 720-by-348 resolution. Either of these graphics modes is initialized automatically and available if you start APL★PLUS II/386 with the graphics character set drivers and the `Env=1024` startup parameter.

To experiment with the graphics functions or to debug a graphics program on a hardware configuration that has only one monitor, start APL★PLUS II/386 with the graphics character set drivers. Once in APL, you can press Alt-F7 to switch between graphics mode and text mode so you do not lose the benefits of a

ROM-generated character set . ⎕*GINIT* is not needed to switch to graphics mode since it is implicitly done during startup.

If the display becomes garbled because you incorrectly used ⎕*GINIT* or 0 ⎕*GINIT* 0, press Ctrl-Esc or Alt-F7 to restore the system to the correct text or graphics mode.

Display Environment

When working in graphics display, you draw on an **easel**. The easel is an imaginary square 1024 by 1024 points in size. The actual display is the projection of this easel onto the screen. The part of the screen that contains the easel is called the **viewport**.

The origin for each screen is in the upper-left corner. The positive X direction is to the right; the positive Y direction is down. The range of the coordinates in each direction is 0 to 1023, inclusive.

The state-setting functions ⎕*GWINDOW* and ⎕*GVIEW* determine how much of the easel is visible and the viewport. ⎕*GWINDOW* specifies the portion of the easel you want to view. ⎕*GVIEW* selects the viewport. The default values project the entire easel onto the entire screen. If part of the graphics display is outside the boundaries of the visible easel, it is clipped and you cannot see it on the screen. Figure 8-1 shows the relationship of the easel and the viewport.

Since most display monitors do not have square screens, the uneven scaling of the display screen coordinates causes pictures to appear distorted, with height and width out of proportion.

For example, consider a monitor that has a display 24 cm wide by 18 cm high, like the IBM color monitor. The horizontal distance between points on the display screen is 0.0234 cm (24÷1024); the vertical distance is 0.0176 cm (18÷1024). This means that the actual distance on the screen between the points whose coordinates are (0,0) and (100,0) is 1.333 (24÷18) times as great as the distance between the points with coordinates (0,0) and (0,100).

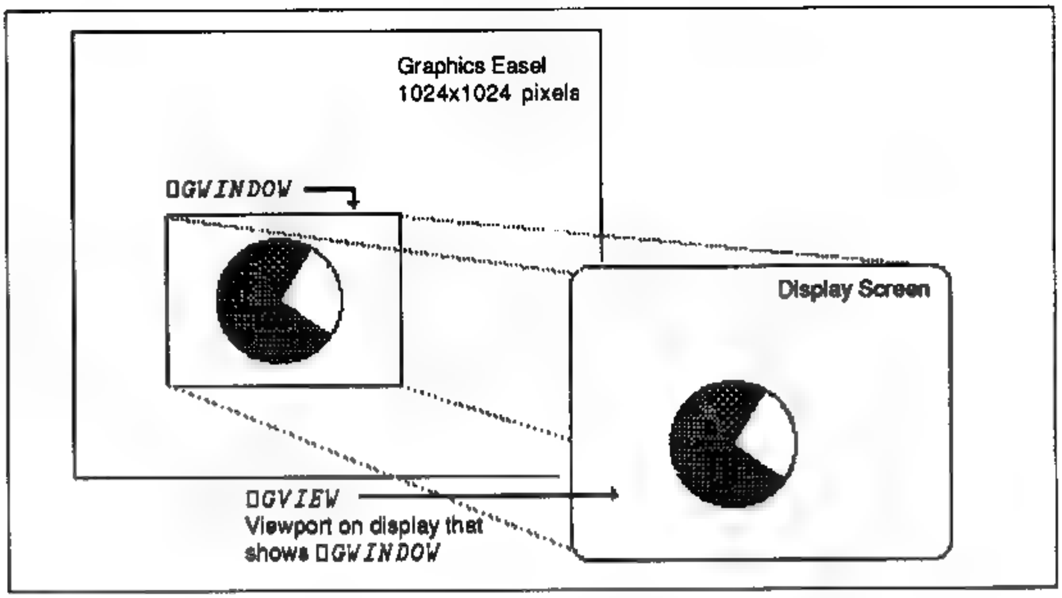


Figure 8-1. Easel and Viewport

Use ⎕*GWINDOW* and ⎕*GVIEW* to alter the dimensions of either screen to control how images from the easel are projected onto the viewport. Appropriate choices for either ⎕*GWINDOW* or ⎕*GVIEW* will reduce distortion by matching the viewport proportions to the selected easel window proportions.

Using Graphics without a Graphics Monitor

If your computer does not have a graphics adapter, you can set aside a portion of memory to hold the graphic image when you start APL. You can then print this image on an EPSON-, Hewlett-Packard LaserJet+-, or DeskJet-compatible printer.

To use the memory-based graphics option, use the `R=` startup parameter to reserve the graphics memory. Follow `R=` with a reservation size to select how much memory you want to set aside for graphics. The amount of memory needed depends on the resolution you want:

- 16 KB for low resolution
- 31 KB for medium resolution
- 89 KB for high resolution
- 128 KB for maximum resolution.

For example, to start APL with external graphics memory, enter:

```
C>apl387 r=31
```

Initialize the external graphics memory with:

option *⎕GINIT 'PRINTER'*

You can use the memory as if it were a second monitor. When the graphics image is complete, you can use *⎕GPRINT* to transfer the image in memory to the printer.

Graphics System Functions

In addition to ⎕*GINIT*, ⎕*GVIEW*, and ⎕*GWINDOW*, there are three other state-setting functions: ⎕*GZOOM*, ⎕*GTYPE*, and ⎕*GMASK*.

- ⎕*GZOOM* allows you to vary the size of text displayed by ⎕*GWRITE* and to magnify pixel patterns when you draw lines and curves with ⎕*GLINE* and ⎕*GCIRCLE*.
- ⎕*GTYPE* allows you to set how displays will mix when one picture overlays another.
- ⎕*GMASK* allows you to hide certain pixels to create dotted or dashed patterns instead of solid lines.

All state-setting functions affect pictures and text drawn after the functions are set. The current graphics display is not affected when you change the settings controlled by any state-setting function (except ⎕*GINIT*, which you can set to clear the screen).

The major picture-drawing graphics function is ⎕*GLINE*. Depending on the rank of its right argument, ⎕*GLINE* allows you to plot individual points, draw solid bars, or trace curves formed by connecting specified points with line segments. You can draw figures in either solid or striped colors.

Use ⎕*GCIRCLE* to draw circles, ellipses, arcs, and pie-shaped wedges in either solid or striped colors.

Use ⎕*GWRITE* to write text at any screen position.

Use ⎕*GSHADE* to fill bars with solid colors or patterns derived from the bit representations of the characters of the graphics character set.

Use ⎕*GPAINT* to paint an area with solid colors or patterns.

Graphics Characters

APL★PLUS II/386 contains two different character fonts. The screen characters used in text display are generated by a program supplied by STSC with your system. There is also an internally stored 8-by-8 character font used for printers that support an EPSON dot-matrix display (t=3) when it prints APL font. When you use the `Env=1024` startup parameter, *⎕GWRITE* uses this internal font to place text on the graphics display and to generate screen characters.

You can use *⎕PEEK* and *⎕POKE* to see the pixel pattern for each character in the graphics display character font, and to substitute characters you define for the characters in the system font. By substituting special characters for some of the characters in the graphics character set, you can produce custom-designed fill patterns to use with *⎕GPAINT* and *⎕GSHADE*.

Each graphics character is represented as a pattern that is eight pixels wide by eight pixels high. An individual character is stored in eight bytes of memory. The first byte contains the bit pattern for the top row of eight pixels, the second byte contains the bit pattern for the next row of eight pixels, and so on.

You can locate the beginning of the graphics character set with:

```
      ⎕SEG←0 ⋄ 256⊥⎕PEEK 177 176
```

Therefore, the eight bytes that represent the rows of the particular character ⎕AV[n+⎕IO] are at locations:

```
      L←(8×n)+256⊥⎕PEEK 177 176
```

You can display them by executing:

```
      ⍉' ⎕'[⎕IO+(8⍴2)⊤⎕PEEK L+(⍳8)-⎕IO]
```

Producing a Hard Copy of a Graphics Screen

You use ⎕*GPRINT* to print the graphics screen image, either as it appears on the screen or as it is held in the computer's memory-resident graphics area created when you use the R= parameter. This technique requires that you define print patterns to represent the various colors and specify the zoom factors (the degrees of magnification the system uses in each dimension) for printing.

If you are generating graphics on both the display and the r= memory, ⎕*GPRINT* refers to the device that was initialized in the most recent ⎕*GINIT*.

Syntax of Graphics System Functions

The system functions that control the graphics environment (⎕*GINIT*, ⎕*GWINDOW*, ⎕*GVIEW*, ⎕*GMASK*, ⎕*GZOOM*, and ⎕*GTYPE*) return their previous setting as a result. To see the current setting, execute these functions using an empty right argument. With the exception of ⎕*GINIT*, these functions do not affect the current screen image; they only affect subsequent displays.

The drawing functions ⎕*GLINE* and ⎕*GCIRCLE* and the filling functions ⎕*GPAINT* and ⎕*GSHADE* have left arguments that describe the color or pattern of the drawing (or fill) and right arguments that specify the location and shape of the figure to be drawn. The drawing and filling functions reshape the left argument, as needed, to match the right argument.

The graphics character writing function ⎕*GWRITE* has a left argument of the character array to be displayed, and a right argument showing the location and color.

Using the VDI Device Drivers

This section explains what VDI graphics device drivers are, how you set up your computer configuration to use them, and how you use the functions in the supplied *VDI* workspace.

What Are VDI Graphics?

The VDI graphics option available in APL★PLUS II/386 involves an interface with external device-independent Virtual Device Interface (VDI) graphics drivers developed and copyrighted by Graphic Software Systems, Inc. (GSS) of Beaverton, Oregon.

VDI is a graphics programming standard. It uses a set of device-independent commands that are translated into device-specific instructions. These instructions allow you to write graphics code that runs on a variety of monitors, printers, and plotters.

GSS*CGI is the Computer Graphics Interface (CGI). This interface links the APL binding and the device-specific drivers and translates commands into low-level instructions.

The *VDI* workspace contains a set of functions that provide the full range of VDI commands, as well as a set of higher level functions such as *BARPLOT* and *PIEPLOT*. These commands outnumber the ⎕*G*-style functions and provide a wider range of capabilities, including interactive graphics and plotter support. For best results, use the Graphic Software Systems' *GSS*CGI Programmer's Guide* in conjunction with the documentation in this chapter.

The set of drivers supplied with APL★PLUS II/386 provides support for most commonly used graphics adapters, printers, and plotters. You can also order additional drivers from STSC. (See the order form in the *Installation Guide*.)

To use the VDI graphics, you must load the appropriate drivers into memory from a CONFIG.SYS or CGI.CFG file before you start APL★PLUS II/386.

The VDI graphics facility uses several layers of cooperating software. First, you must load a device driver. This device driver specifies the output device (graphics adapter, printer, or plotter). To reserve DOS memory, use the `cgisize=` startup parameter. Then, use the functions in the supplied *VDI* workspace to pass instructions to the graphics driver.

(**Note:** The APLCGI.EXE file required for you to use VDI graphics in the APL★PLUS System for the PC is **not** required by APL★PLUS II/386, because the facility provided by APLCGI is built into the APL interpreter.)

You can use VDI graphics from APL with the unlocked functions in the *VDI* workspace.

Loading Device Drivers

There are two ways you can load VDI graphics device drivers. For complete details on both methods, see the *Installation Guide*. In the first method, you place `device=` statements in a CONFIG.SYS file.

To load device drivers from the CONFIG.SYS file, follow these steps **before** starting APL★PLUS II/386.

Warning: Do not follow these steps from within APL; they can cause your system to crash.

1. Place an entry in the CONFIG.SYS file for each device you want to use.
2. If you will be using any "software" fonts, add a line for the font driver device.
3. After the specific entries for the device drivers, add a line for the GSS*CGI driver, GSSCGI.SYS.

In the second method, you place `driver=` statements in a special VDI graphics configuration file named CGI.CFG. This method requires GSS Version 2.14 or later. To load device drivers from a CGI.CFG configuration file, follow these steps.

1. Create an ASCII text file named CGI.CFG using any DOS text editor.

2. Place an entry in the file for each device you want to use.

3. At some point in the file, add a line for the GSS*CGI driver, GSSCGI.SYS.

Once you have set up the CONFIG.SYS or CGI.CFG file properly, follow these steps to initiate VDI graphics:

1. If you used the `/t` parameter with the GSSCGI device statement in the CONFIG.SYS or CGI.CFG files, you must run

   ```
   drivers
   ```

   from DOS. This routine installs the drivers in memory. If you did not use the `/t` parameter, this step is unnecessary. If you use the `/t` parameter in a CGI.CFG file, you must run DRIVERS twice.

2. Start APL★PLUS II/386. You can now run the functions in the *VDI* workspace to generate graphics on the devices whose drivers you have loaded.

3. If you use the `/t` parameter and if the DRIVERS routine was the last terminate-and-stay-resident (TSR) program you ran, you can reclaim most of the memory used by the drivers after you leave APL★PLUS II/386without rebooting your computer. Enter

   ```
   DRIVERS /r
   ```

   from DOS after leaving APL★PLUS II/386.

If you use CGI.CFG, you can reclaim all the memory used by the drivers after you leave APL by entering:

```
DRIVERS /a
```

The file DRIVERS.EXE lets you control the loading and removing of the device drivers. You can run DRIVERS from an AUTOEXEC.BAT file or from a batch file before a graphics application to load GSS*CGI, and after the application to remove GSS*CGI.

You can use the following switches when you call DRIVERS:

- **/a remove all**
  Completely remove GSS*CGI configuration from memory.
- **/f force**
  Force the action of /a or /r.
- **/p presence**
  Display the form of GSS*CGI.
- **/q quiet**
  Do not display informational messages.
- **/r remove**
  Remove transient GSS device drivers from memory.
- **/u update**
  Update the GSS*CGI information.
- **/h display help message**
  Display a help message.

If you run DRIVERS with no switches, it behaves as follows:

- if GSS*CGI is present, it is reinitialized
- if GSS*CGI is not present, it is loaded
- if GSS*CGI is present but transient drivers are not, the drivers are loaded.

Using the *VDI* Workspace

This section is an overview of the functions available in the *VDI* workspace. See the "VDI Graphics Utilities" section in the *Utilities Manual* and the *GSS Programmer's Guide* for detailed information.

The *VDI* workspace contains three types of functions.

- Functions that start with *V_* are cover functions for individual VDI commands.
- Functions whose names begin with *EXAMPLE* are self-contained examples that you can run from immediate execution mode. Those functions that end with a digit send output to the display screen; those that end with a *P* send output to the printer. You can use these functions as models.
- The remaining utility functions perform useful tasks to help you write graphics applications.

The workspace also includes several commented sample functions that direct graphics output to the display screen. Examine these functions carefully before you try to write your own.

If you use a one-monitor system and direct graphics output to the screen, you must call all cover functions from within an APL function. This function must open a workstation, execute the graphics functions, and close the workstation without any interaction with the text screen. No quad (⎕), quote-quad (⍞), implicit output, or return to immediate execution mode is permissible before you close the workstation and return to text mode.

Warning: You may have to reboot your computer if you fail to return to text mode before you return to immediate execution mode. Press Ctrl-Esc to return to the right mode and then close the workstation if it is open.

Follow these steps to display graphics output on an output device and then return to immediate execution mode.

1. Switch to the DOS keyboard driver if any input is expected.
2. Open a workstation with the graphics display, set of default colors, and resolution that you want.
3. Execute the desired graphics functions.
4. Return to text mode.
5. Close the workstation.
6. Switch back to the APL keyboard driver.

Note: You must use the DOS keyboard if you request input from one of the VDI commands. You can switch into or out of the DOS keyboard with the Alt-F5 key.

The following APL function is an example; also see the *DISPLAY* and *PRINT* functions.

```
      ∇ EXAMPLE;S;v_DISPLAY_out;v_INPUT;
v_OUTPUT;v_rc;⎕ELX;⎕ALX
[1]
[2]   ⍝ If anything goes wrong, display error and
[3]   ⍝ return from graphics mode cleanly
[4]
[5]    ⎕ALX←⎕ELX←'CLOSEDISPLAY ⋄ ⎕DM'
[6]
[7]    DOSKB ⍝ DOS keyboard if input expected
[8]
[9]   ⍝ Open display device
[10]   'DISPLAY' V_OPEN_WKST 0 1 1 2 1 1 1 2 6 1 1
[11]  ⍝ The preceding parameters mean
[12]  ⍝ 0 = Scale 32767×32767 coordinates to
        actual screen size
[13]  ⍝ 1 = solid line
[14]  ⍝ 1 = polyline color 1
[15]  ⍝ 2 = use + as polymarker symbol
[16]  ⍝ 1 = polymarker color 1
[17]  ⍝ 1 = graphics text font index
[18]  ⍝ 1 = graphics text color index
[19]  ⍝ 2 = fill interior style='pattern'
[20]  ⍝ 6 = fill style index
[21]  ⍝ 1 = fill color index
[22]  ⍝ 1 = prompting flag='display'
[23]
[24]   v_INPUT← v_OUTPUT← v_DISPLAY ⍝ Output
       device is 'DISPLAY'
[25]
[26]  ⍝ Draw filled circle at left-center
[27]   V_CIRCLE 8192 16384 4096
[28]  ⍝ Change fill style index from 6 to 14
[29]   V_SET_FILL_STYLE_IND 14
[30]  ⍝ Draw filled circle at right-center
[31]   V_CIRCLE 24576 16384 4096
[32]  ⍝ Draw title at top
[33]   16000 30000 V_GRPH_TXT 'Circles'
[34]  ⍝ Wait for user input before ending display
[35]   S←V_REQ_CHOICE 0
[36]
[37]   V_ENT_CUR_MODE ⋄ V_CLOSE_WKST v_DISPLAY
[38]  ⍝ Return to APL keyboard mode
[39]   APLKB
    ∇
```

9 Communications

This chapter tells you how to use your computer as a terminal to a remote computer and how to transfer information between your computer and other computers or peripheral devices. The chapter is organized as follows.

- "Using Terminal Mode" explains how to use your computer as a terminal connected to a remote computer.
- "Communicating with Remote Devices" describes the system function ⎕*ARBIN* and explains how you use it to set up and monitor data exchange, create print files, and gather information from the keyboard.
- "Transferring Data from Other APL Systems" is an overview of the techniques you can use to transfer data among different APL implementations.

Using Terminal Mode

When you use your computer as a terminal to a remote computer, you are using terminal mode. Your computer must have an RS-232 port or a built-in modem. You do not need any other communications software.

Once you activate terminal mode, you can switch between terminal mode and local mode whenever you want. The input and output of terminal mode is recorded in the APL session.

The system emulates two kinds of terminals:

- Datamedia 1520 (the default)
- VT100 (ANSI standard terminal).

Check with the remote system's operator or administrator to determine which emualtion you should use. STSC has enhanced both modes of terminal emulation, since neither can display both APL and ASCII characters at the same time. Therefore, you should not use the information in Tables 9-2 and 9-3 to write software for the actual Datamedia 1520 or DEC VT100 terminals.

To switch from Datamedia emulation to VT100 emulation, you can do any of the following.

- Place a line in the configuration file of the form:

  ```
          termstyle=vt100
  ```

  The next time you start APL★PLUS II/386, the system automatically emulates the VT100 terminal.

- Select the emulation you want from the Style Menu after you activate terminal mode. (See the "How to Enter and Leave Terminal Mode" section later in this chapter.)

- Set ⎕*SEG*←⊖. Execute 1 ⎕*POKE* 315 to switch from Datamedia to VT100 emulation. Execute 0 ⎕*POKE* 315 to switch from VT100 to Datamedia emulation.

How to Enter and Leave Terminal Mode

You can activate terminal mode in three ways:

- press Alt-F8
- select Terminal Mode from the Open Menu
- execute `1 ⎕POKE 117` (useful under program control).

The words "Terminal Mode" and several related state settings appear on the status line once you activate terminal mode.

Once you begin your session, the APL★PLUS II/386 interpreter and the contents of your active workspace are disconnected from your personal computer's keyboard and screen. Anything you type is sent through the communications port and any data from the remote computer is sent through the communications port and displayed on your screen.

You must follow the conventions and prompts of the remote system you are using. APL★PLUS II/386 does no more than display and transmit your input, and receive and display the remote computer's output.

Using Terminal Mode Menus

Terminal mode has its own special set of menus, which you can use to change or set up your communications parameters before you actually sign on to the remote system. Display them the same way you display the menus in local mode: press Ctrl-/. You see the menu bar in Figure 9-1.

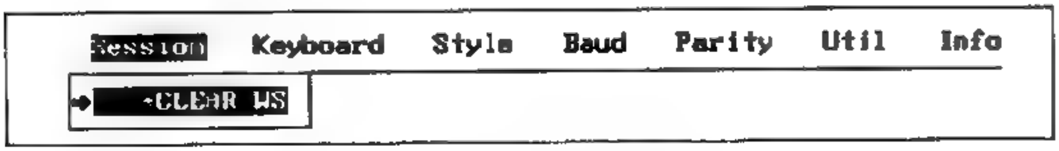


Figure 9-1. Terminal Mode Menus

The Session, Keyboard, Util, and Info menus are the same as those for the session manager (see the Editing with the Session Manager chapter in this manual).

The Style Menu

The Style menu, shown in Figure 9-2, lets you set up your terminal session to suit your needs.

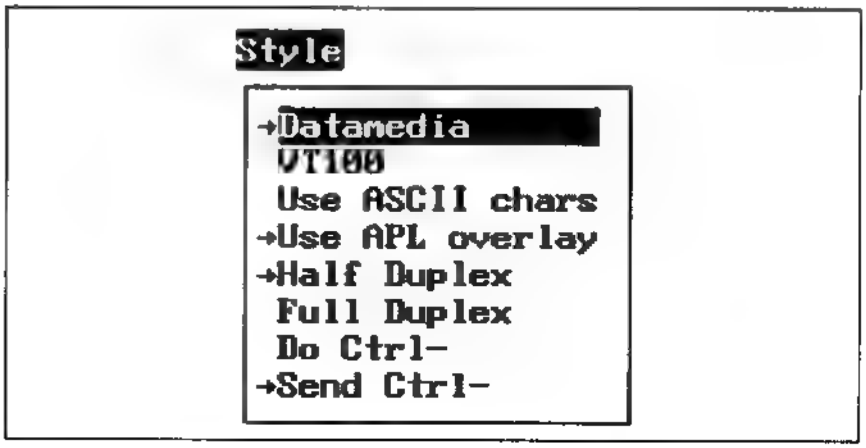


Figure 9-2. The Style Menu

The Style menu options are:

- **Datamedia/VT100**
  Use this option to select between Datamedia or VT100 terminal emulation.

- **Use ASCII chars/Use APL overlay**
  Use this option to select either an APL or ASCII character set for your terminal session. If you select the APL character set, your computer acts as a typewriter-paired APL-ASCII terminal.

- **Half Duplex/Full Duplex**
  Use this option to specify whether you want to use full or half duplex. Half duplex echoes what you type; full duplex does not.

- **Do Ctrl/Send Ctrl**
  Use this option to specify whether you want certain Ctrl- keys to perform local actions or send the appropriate codes to the remote computer; for example, whether Ctrl-B will break a line at the cursor or transmit ASCII code 2.

The Baud Menu

The Baud menu, shown in Figure 9-3, lets you select the baud rate for your terminal session. Baud rate controls the speed of data transmission. The default is the value last set by the DOS MODE command. If you have not used the MODE command, DOS sets the baud rate to 2400.

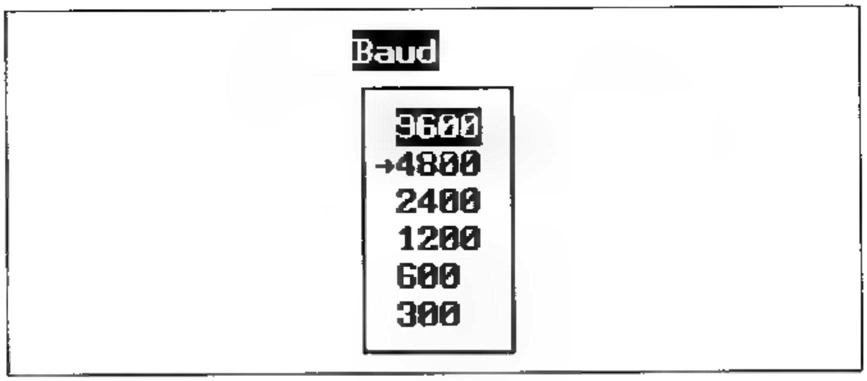


Figure 9-3. The Baud Menu

The Parity Menu

The Parity Menu, shown in Figure 9-4, allows you to set the parity, databits, and stopbits for correct data transmission. The settings depend on what the remote computer expects. Databits 8 is inconsistent with anything other than No Parity.

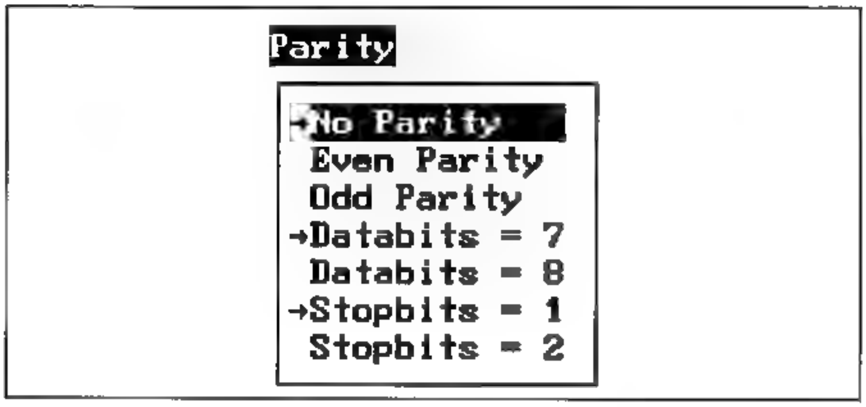


Figure 9-4. The Parity Menu

Using Terminal Character Sets

When you use terminal mode, you can send your keystrokes to the remote system as either ASCII characters or as an APL overlay that sends APL characters.

You can specify how to transmit characters in three ways:

- in the configuration file with the `termmode=` parameter
- by selecting the appropriate option from the Style menu
- by pressing Ctrl-CapsLock to toggle the current state.

The character transmission, ASCII or APL overlay, shows in the status line. In APL★PLUS II/386, the character transmission mode (ASCII or APL overlay) is separate from the keyboard layout (Text or APL). To change both keyboard layout and the transmission mode, press both Ctrl-CapsLock to change the transmission mode, and Alt-CapsLock to change the keyboard type.

Exception: If the system is in ASCII transmission mode and in terminal mode, the keyboard always functions as a text keyboard, regardless of the underlying keyboard state and no matter how you were placed into ASCII transmission mode.

When sending ASCII characters, APL★PLUS II/386 emulates a terminal with the standard uppercase and lowercase character set used by all ASCII terminals. Pressing a key sends the character that would be displayed using the text keyboard layout, regardless of the current keyboard state. For example, pressing a shifted A sends an ASCII uppercase A; in half duplex, an uppercase A is displayed. Use this option when accessing non-APL programs on remote computers.

When sending characters using the APL overlay, APL★PLUS II/386 emulates a standard APL terminal using the typewriter-pairing transmission codes. Pressing a key will send a different character depending on whether the APL or text keyboard is in use. For example, pressing a shifted A sends the APL overlay of alpha (⍺) (an ASCII uppercase A) in the APL keyboard, or the APL overlay for the APL uppercase A; ASCII lowercase a (*a*) in the text keyboard. Use this option when you run APL on the remote computer.

You see the usual wrap marker for lines longer than the window in terminal mode as well as in local mode.

How Your Computer Transmits Information

Table 9-1 shows what your computer transmits when you press certain keys. These events are independent of the main keyboard tables. The columns list the keystrokes sent and the behavior of the two emulators.

Note the following when you use terminal mode:

- You can use the Ctrl key in combination with a character key to send other control codes. In these cases, APL★PLUS II/386 transmits the control character that would be sent by an ASCII terminal. To send control codes in this manner, you must be in "Send Ctrl-" mode (the default). Use the Style menu to choose this mode.
- The system sends characters one at a time as you type them. The system displays printing characters on the screen at the current cursor location, unless you are in full duplex mode.
- The system sends lowercase characters as regular lowercase characters when you use the ASCII (text) transmission mode. They are sent as underlined APL letters when you use the APL overlay mode. For example, *a* is transmitted as three ASCII characters (*A*, BS, and _), corresponding to what would be transmitted by an APL/ASCII terminal when you form the overstrike for *A̲*.
- When you use the APL overlay mode, a keystroke that generates a composite (overstrike) character, such as ⍎, transmits the composite as an overstrike; for example, ⊥,⎕*TCBS*,∘.
- When you press Shift-Alt-Home to transmit an ASCII FF in half duplex, the window clears and the cursor moves to the upper-left corner.

Table 9-1. Transmitted Keystrokes

	Behavior	
Keystrokes Sent	Datamedia	VT100
Enter	Return–ASCII CR	Return–ASCII CR
Alt-Enter	Transmit entire line	Transmit entire line
Ctrl-Break	ASCII BREAK	ASCII BREAK
Up (↑)	ASCII US–Decimal 31	Esc [A
Down (↓)	ASCII LF (Linefeed)	Esc [B
Right (→)	ASCII FS–Decimal 28	Esc [C
Left (←)	ASCII BS (Backspace)	Esc [D
Ctrl-↑	Four ASCII US	Four Esc [A
Ctrl-↓	Four ASCII LF	Four Esc [B
Ctrl-→	Eight ASCII FS	Eight Esc [C
Ctrl-←	Eight ASCII BS	Eight Esc [D
Shift-Alt-→	ASCII GS–Decimal 29	ASCII GS–Decimal 29 (Ctrl-])
Tab	ASCII HT	ASCII HT
Esc	ASCII ESC	ASCII ESC
Shift-Alt-Home	ASCII FF	ASCII FF
Del	ASCII DEL	ASCII DEL
Ctrl-N	ASCII SO–change transmission type to APL overlay	ASCII SO–change transmission type to APL overlay
Ctrl-O	ASCII SI–change transmission type to ASCII (see **Exception**)	ASCII SI–change transmission type to ASCII (see **Exception**)
Ctrl-H	ASCII BS–Decimal 8	ASCII BS–Decimal 8
Ctrl-J	ASCII LF–Decimal 10	ASCII LF–Decimal 10
Ins	ASCII CTRL A	ASCII CTRL A
Backspace	ASCII BS LF	ASCII BS LF

How Your Computer Behaves when It Receives Information

Terminal mode recognizes two types of information sent from a remote computer: control sequences and escape sequences.

Control sequences consist of a byte whose decimal value falls between 0 and 31, followed by zero or more additional bytes. Control sequences cause different behavior depending on the emulator you choose. Table 9-2 shows the control sequences and their behavior in both emulators (assuming ⎕*IO*←0), except those beginning with Esc (decimal 27), which are treated separately later.

Escape sequences consist of a decimal 27 (ASCII ESC or ⎕*TCESC*), followed by other bytes. These items are the same in both emulators; terminal mode responds to all of them in the same way. Table 9-3 list the sequences. In the table, *Pn* represents number of characters.

Table 9-2. Control Sequences and Behavior

Control Sequence		Behavior	
ASCII	Dec.	Datamedia	VT100
BEL	7	chirp	chirp
BS	8	nondestructive backspace	nondestructive backspace
HT	9	tab	tab
LF	10	linefeed	linefeed
VT	11	erase to end of window	vertical tab
FF	12	clear window	linefeed
CR	13	carriage return	carriage return
SO	14	select APL overlay transmission*	select APL overlay transmission*
SI	15	select ASCII transmission*	select ASCII transmission*
CAN	24	ignored	cancel escape
EM	25	cursor to upper-left corner	ignored
SUB	26	ignored	cancel escape
FS	28	cursor right one column	ignored
GS	29	erase to end of line	ignored
RS	30	cursor movement**	ignored
US	31	cursor up one line	ignored

*See **Exception** noted earlier.
**Receiving an ASCII RS (decimal 30) causes the next two characters received to be treated as cursor movement to the column and row specified by subtracting 31 from the decimal value of the two characters.

Table 9-3. Recognized Escape Sequences

Escape Sequence	Decimal Value	Behavior
Esc t	27 116	Exits terminal mode.
Esc x	27 120	Exits terminal mode while capturing the line sent by the host and executing it. Expressions to be executed in local mode should use quote-quad assignment for proper display on the host, so the cursor stays on the line to be executed.
Esc [P	27 91 80	Deletes the character at the cursor and closes up the space. Same as Del in local mode.
Esc [M	27 91 77	Deletes the line containing the cursor and closes up the gap. Same as Alt-F4 in local mode.
Esc [L	27 91 76	Inserts a blank line at the cursor. Same as Alt-F3 in local mode.
Esc S 0	27 83 48	Disables the off-screen scrolling capability.
Esc S 1	27 83 49	Enables the off-screen scrolling capability.
Esc a *n*	27 97 64–79	Sets the default display attribute. Valid values are 64 plus the sum of the values of the desired attributes: 1=reverse video 2=high intensity 4=blink 8=underline
Esc [4 h	27 91 52 104	Switches to insert mode.
Esc [4 l	27 91 52 108	Switches to replace mode.
Esc [*Pn* A		Moves cursor up *Pn* times. The cursor stops at the top line without beeping. If *Pn* is omitted, 1 is assumed.
Esc [*Pn* B		Moves cursor down *Pn* times. Stops at the bottom of the window without beeping.
Esc [*Pn* C		Moves cursor right *Pn* times. Stops at the right margin.
Esc [*Pn* D		Moves cursor left *Pn* times. Stops at the left margin.

Table 9-3. Continued

Escape Sequence	Decimal Value	Behavior
Esc [*Pl* ; *Pc* H		Positions cursor at line *Pl*, column *Pc*. Missing parameters are assumed to be 1.
Esc [H		Moves cursor to line 1, column 1 (Home).
Esc [*Pl* ; *Pc* f		Positions cursor. Same as Esc [H.
Esc D		Moves cursor down.
Esc M		Moves cursor up.
Esc E		Moves cursor to left margin of next line. Scrolls up at bottom of window.
Esc 7		Saves cursor, attribute, APL/ASCII transmission state.
Esc 8		Restores cursor, attribute, APL/ASCII transmission state saved with last Esc 7. If Esc 7 was not used, cursor moves to Home.
Esc [K		Erases to end of line.
Esc [0 K		Erases to end of line.
Esc [1 K		Erases from start of line to cursor.
Esc [2 K		Erases entire line.
Esc [J		Erases to end of window.
Esc [0 J		Erases to end of window.
Esc [1 J		Erases from start of window to cursor.
Esc [2 J		Erases entire window.
Esc [*Pn* P		Deletes *Pn* characters. If *Pn* is omitted, 1 is assumed.
Esc [*Pn* L		Inserts *Pn* lines. If *Pn* is omitted, 1 is assumed.
Esc [*Pn* M		Deletes *Pn* lines. If *Pn* is omitted, 1 is assumed.
Esc [? 5 i		Turns on slaved printer.
Esc [5 i		Turns on printer controller (output only to printer).
Esc [? 4 i		Turns off slaved printer.
Esc [4 i		Turns off printer controller.

Table 9-3. Continued

Escape Sequence	Decimal Value	Behavior
Esc [i		Prints screen. Same as Shift-PrtSc.
Esc [0 i		Prints screen.
Esc [0 m		Selects normal screen attribute. Similar to Esc a *n*.
Esc [m		Selects normal screen attribute.
Esc [1 m		Selects high intensity screen attribute.
Esc [4 m		Selects underline.
Esc [5 m		Selects blink.
Esc [7 m		Select reverse video.
Esc [*Pt* ; *Pb* r		Sets top and bottom of window. Changes the first and third values of ⎕*WINDOW*. Values are in origin 1. If either parameter is omitted, 1 is assumed for the top and 24 is assumed for the bottom.
Esc [4 h		Turns on insert mode.
Esc [4 l		Turns off insert mode.
Esc [? 7 h		Turns on wrap.
Esc [? 7 l		Turns off wrap; when a character is typed in the rightmost column, the cursor does not advance.
Esc [*Pn* z		Prints element *Pn* of ⎕*AV*.
Esc H		Sets tab stop at current cursor column.
Esc [g		Clears tab stop at current cursor column.
Esc [0 g		Clears tab stop at current cursor column.
Esc [3 g		Clears all tab stops.

The system uses the following reports from terminal to remote:

- **Device Status**
  The remote computer sends ⎕*TCESC*,'[5*n*'. The terminal responds ⎕*TCESC*,'[0*n*' (*terminal OK*).

- **Printer Status**
  The remote computer sends ⎕*TCESC*,'[15*n*'. The terminal responds ⎕*TCESC*,'[?10*n*' (*printer ready*) or ⎕*TCESC*,'[?11*n*' (*printer not ready*).

- **Device Attribute:**
  The remote computer sends ⎕*TCESC*,'[c' or ⎕*TCESC* '[0c'. If VT100 emulation is in effect, the terminal responds ⎕*TCESC*,'[?6c'. If VT100 emulation is not in effect, the terminal does not respond.

- **Cursor Position**
  The remote computer sends ⎕*TCESC*,'[6*n*'. The terminal responds ⎕*TCESC*,'[*Pl*;*PcR*, where *Pl* is line, *Pc* is column; for example, ⎕*TCESC*,'[12;34*R*'. Lines are numbered in origin 1.

Customizing Control and Editing Keys

The terminal mode transmission table controls the local action and sequence of characters that the system transmits you press a key. This feature is useful if you use terminal mode to sign on APL★PLUS II/UNIX or APL★PLUS for VAX/VMS, because you can program the special keys to produce the same effect in terminal mode as they do when you use APL★PLUS II/386 by itself.

For details on customizing these tables, see the Advanced Techniques chapter in this manual.

Communicating with Remote Devices

The system function ⎕*ARBIN* provides a facility for detailed control of input and output between APL★PLUS II/386 and another computer connected to a serial port. You can use ⎕*ARBIN* as the basis for transferring data between the two computers under program control. The serial transfer and IRMA transfer facilities are examples of this kind of data transfer.

The syntax for ⎕*ARBIN* is

result ← *paramlist* ⎕*ARBIN* *prompt*

where *result* contains any received data, *paramlist* is a list of parameters that describes where and how to communicate, and *prompt* is the data to be transmitted.

The left argument is a vector or singleton of integers describing how to carry out the communication. If the left argument is a singleton, it specifies the destination of the data and does not wait for input, but immediately resumes local processing. If input is expected, or if you specify values other than the default, the left argument contains the following elements:

[1]	outport — destination of data
[2]	inport — source from which data is to be received
[3]	translation — translation table to be used
[4]	protocol — transmission protocol to be used
[5]	wait — number of seconds to wait for data
[6]	charlimit — maximum number of characters of data
[7]	terminators — list of termination codes.

The ⎕*ARBIN* function uses the elements starting from the left; if you specify fewer than six elements, the remaining parameters use the default values. For detailed information, see ⎕*ARBIN* in the "System Functions, Variables, and Constants" chapter of the *Reference Manual*.

Setting the Ports

The first two elements of the parameter list tell □*ARBIN* the destination of the data it is sending (the outport) and the source of the data it is receiving (the inport).

The possible ports are shown in the following list.

0	No port
1	First serial port (COM1)
2	Second serial port (COM2)
3	First parallel port (LPT1, outport only)
4	Second parallel port (LPT2; outport only)
5	Third parallel port (LPT3; outport only)
6	Screen characters or actions starting at cursor
7	Screen attributes starting at cursor
8	Insert screen characters starting at cursor (outport only)
9	Screen actions starting at cursor (outport only)
10	User-supplied driver routine
11	Keyboard
12	DOS standard input
13	DOS standard output
14	DOS standard auxiliary
15	DOS standard printer
16	Get character
17	Get character or event
18	Get event
19	Get raw keystroke

Depending on the outport or inport selected, other elements of the left argument can become meaningless. Meaningless elements are ignored.

- **Using the Pseudo-Port — Port 0**
  Port 0 is a pseudo-port used to send data specifying that no response is desired or to receive data without transmitting anything in return. This port is used in conjunction with the other ports; examples of its use are shown in the following sections.

- **Using Serial Ports — Ports 1 and 2**

  Ports 1 and 2 are the first and second serial ports, also known as COM1 and COM2 to DOS. They are used to communicate with devices such as modems, remote computers, plotters, or serial printers. If you use *⎕ARBIN* with these ports, you must set the baud rate with the DOS MODE command, or select it from the Baud menu for Port 1. These ports send and receive data, allowing you to set up direct communication between a remote computer or other device and your computer.

  For example, if you have a modem hooked up to your first serial port, you can send the following command to tell a Hayes or compatible modem to dial the telephone:

  ```
        1 0 2 ⎕ARBIN 'ATDT5551212', ⎕TCNL
  ```

  This command has the same effect as pressing Alt-F8 to enter terminal mode and typing *ATDT*5551212 and pressing Return.

  If you have an APL★PLUS System connected in terminal mode, you can send a function to a remote mainframe with the command:

  ```
        1 ⎕ARBIN ⎕VR 'FOO'
  ```

  The *⎕ARBIN* function transmits the function *FOO* to the mainframe. (**Note**: If the remote APL★PLUS System is APL★PLUS II/UNIX or APL★PLUS for VAX/VMS, that system must be in overstrike mode to receive composite characters sent by *⎕ARBIN*.) To bring a function from a mainframe to your computer, use:

  ```
        1 1 0 0 ¯1 32767 7 ⎕ARBIN
  '⎕VR''FOO''',⎕TCNL
  ```

  The *⎕ARBIN* function brings the character representation of the function to the APL★PLUS System.

  To use these two serial ports, you must be sure that you know the correct communications protocol. Different types of hardware recognize different types of protocol. If you specify them incorrectly you could send or receive incorrect data.

- **Using Parallel Ports — Ports 3, 4, and 5**
  These ports refer to the three parallel ports, known as LPT1, LPT2, and LPT3 to DOS. You typically use them to connect a printer to your computer. You can use parallel ports only to transmit data; you cannot receive data through them. Use the *translation* parameter to determine how a printer reads and prints the data you send.

  For example, to send an APL program listing to an EPSON printer on LPT1, use:

  ```
          3  0  3 ⎕ARBIN ⎕VR 'fnname'
  ```

  Note: APL printer support often uses Port 10; see the Displaying and Printing APL Characters chapter in the *Utilities Manual*.

  Many printers allow you to send control codes or escape sequences to modify settings (such as the print mode, margins, line spacing, and so on) and to choose a character set. You typically set these codes by sending an ASCII Escape code followed by a series of characters. The Escape code is stored in the variable ⎕TCESC. For example, the sequence Esc- might send start or end underlining. To send this command, you use:

  ```
          3 ⎕ARBIN ⎕TCESC, '-'
  ```

  Check your printer's manual for the appropriate escape codes.

- **Writing Characters to the Screen — Ports 6, 7, and 8**
  These ports allow you to write characters or attributes to the screen starting at the position of the cursor. The system translates all nonprintable characters to a visible character. Ports 6 and 8 send characters; Port 6 overwrites whatever is on the screen; Port 8 inserts the characters ahead of whatever is on the screen. Port 6 also allows you to specify actions such as cursor movements; for example, to move the cursor up 10 lines, you could execute

  ```
          6 ⎕ARBIN 10⍴393
  ```

  where 393 is the event number for up arrow (↑).

Port 7 changes screen attributes. Try the next example to place six spaces of reverse video on your screen. (The example uses ⎕*PEEK* 170, the highlight attribute, to generalize reverse video, and assumes that ⎕*SEG* is empty.)

```
        7 ⎕ARBIN 6⍴⎕PEEK 170
```

You could use this highlighted field to take data input from a user by repositioning the cursor at the beginning of the field.

- **Controlling Screen Actions — Port 9**
  Port 6 now supersedes Port 9, but it is included for compatibility with other APL★PLUS systems. Port 9 controls screen actions like cursor movement, scrolling, and deletion. The right argument contains the numeric key values that represent the movements. (Use 256|*keyvalues*, listed in the Atomic Vector and Keyboard Events appendix in the *Reference Manual*.)

  For example, move the cursor up 10 lines with the command

```
        9 ⎕ARBIN 10⍴137
```

  where 137 (=256|393) represents the up arrow (↑ ^ key.

- **Using Custom Driver Routines — Port 10**
  This special port points to a user-supplied driver routine using interrupt 90 hex. For example, APLPRINT.COM and APLPS.COM (supplied with the system) are assembler routines that drive this port to print APL characters on a printer that supports a downloaded character set. All printer support in the *PRINTERS* workspace uses this port.

- **Reading Characters from the Keyboard — Port 11**
  This port, used as an inport, reads characters from the event buffer of the keyboard. The characters can be returned in the result as either the actual character or the numeric value represented by the key. The *translation* parameter changes the form of the result. A 2 returns the character; a ¯1 returns the numeric value.

  Other parameters that affect this port are *wait, charlimit,* and *terminators*. For example, you could read and assign the actual characters that a user types, up to a maximum of 10 characters, wait no longer than 20 seconds for them to be typed, and terminate the sequence if the user presses Enter.

```
      A ← 0 11 2 0 20 11 13 ⎕ARBIN ''
```

The *charlimit* is set to 11 since the last character is always the terminator.

Another use of Port 11 is to emulate the BASIC language INKEY$ function. INKEY$ does not wait for a character to be placed in the input buffer, as ⎕*INKEY* does. This behavior can be accomplished in APL with the statement:

```
      ¯1↓ 0 11 2 0 0 2 ⎕ARBIN ''
```

The ¯1↓ removes the terminator, while the wait of 0 specifies what is there without waiting to see if the event buffer is empty.

- **Using Standard DOS Devices — Ports 12, 13, 14, and 15**
  These ports specify the DOS "standard" devices, input, output, auxiliary, and printer. A typical use for these ports might be to send output to another device or file using the DOS redirection capability.

- **Other Keyboard Ports — Ports 16, 17, 18, and 19**
  Port 16 is similar to Port 11, except that only keystrokes valued less than 256 (that is, elements of ⎕*AV*) are included in the result. Other keystrokes are performed (echoed) if possible.

  Port 17 is equivalent to Port 11. Keystrokes valued less than 512 are included in the result. Other keystrokes are performed if possible.

  Port 18 is similar to Port 11, but all keystrokes except immediate actions, including keys such as CapsLock, are included in the result.

  Port 19 returns raw keystrokes. The values returned are 1000 times the physically typed shift state plus the keybutton number. The shift state indicates if Shift, Ctrl, or Alt has been pressed. The keybutton number is the value of the key.

See the Advanced Techniques chapter in this manual for more information. Events in the ⎕*INBUF* buffer are ignored.

A negative number between ¯32768 and ¯1 found in ⎕*NNUMS* can also be used as either an outport or inport. These numbers access the native file or device tied to that number. When one is used as an outport, ⎕*ARBIN* reads from that device or file starting at the current pointer position (see ⎕*NREAD* in the System Functions, Variables, and Constants chapter in the *Reference Manual*).

Transferring Data from Other APL Systems

APL★PLUS II/386 contains three transfer facilities to help you exchange workspaces and files with APL systems on other computers:

- The serial transfer facility (*SERXFER*) contains workspaces for transferring APL data and files between APL★PLUS II/386 and a host APL system through a serial port. You can also use *SERXFER* to move workspaces, since it contains facilities used to store function and variable definitions in APL component files, and use these definitions later to re-create the objects.
- The IRMA transfer facility (*IRMAXFER*) contains workspaces for transferring APL data and files between APL★PLUS II/386 and a host APL system through an IRMA board. You can also use *IRMAXFER* to move workspaces, since it contains facilities used to store function and variable definitions in APL component files, and use these definitions later to re-create the objects.
- The Source Level Transfer facility (*SLT*) can migrate workspaces and files from one APL system to another using the Workspace Interchange Convention (WIC). *SLT* works by building specially formatted binary files that contain a character set definition and representations of functions and variables. You then move this file is then moved to a destination system. You use an *SLT* facility on the destination system to combine the file with the system character set definition, form a translate table, and then reconstitute the original functions and variables.

Each of these facilities is explained in detail in the Transferring Data chapter in the *Utilities Manual.*

10 Exception Handling

The exception handling facility implemented in APL★PLUS II/386 allows APL functions to react directly to errors and certain other events. This chapter explains how you use the exception handling facilities of APL★PLUS II/386 and is organized as follows.

- "Principles of Exception Handling" discusses the concepts underlying exception handling and the system features for exception handling.
- "Basic Algorithms for Exception Handling" explains how to use the system features to build fundamental algorithms for exception handling.
- "Using the *HANDLERS* Workspace" explains how you can use the functions in the supplied *HANDLERS* workspace to build exception handling into your programs.

Exception handling has two main benefits: reliability and security.

- **Reliability.**
  Exception handling makes it possible for you to correct many common execution-time errors, and to react to them in a controlled fashion, protecting your application users from the consequences of their mistakes.
- **Security.**
  Using exception handling, you can ensure that application programs, and the data they use and maintain, are not susceptible to accidental or malicious interference.

Principles of Exception Handling

The exception handling facility comprises three system variables (*⎕ALX*, *⎕ELX*, and *⎕SA*), two system functions (*⎕ERROR* and *⎕DM*), and a workspace (*HANDLERS*).

- ***⎕ALX* — Attention Latent Expression.**
  This system variable is a latent expression executed in the event of a weak interrupt.
- ***⎕DM* — Diagnostic Message.**
  This system function returns the last diagnostic message recorded in the workspace.
- ***⎕ELX* — Error Latent Expression.**
  This system variable is a latent expression executed in the event of certain errors.
- ***⎕SA* — Stop Action.**
  This system variable specifies a "last resort" action taken if a task enters immediate execution mode.
- ***⎕ERROR* — Error Signaling.**
  This system function removes the function executing from the execution stack and signals an error in the environment.

Errors and Attentions

While a task is executing in APL★PLUS II/386, certain events can occur that prevent completion of the task. These events fall into two broad classes: errors and attentions.

- **Error**
  An error results when an APL statement or system command is ill-formed or when the environmental conditions necessary to complete it are not met. Examples of these errors are *WS NOT FOUND*, *SYNTAX ERROR*, *INCORRECT COMMAND*, and *DOMAIN ERROR*.

- **Attention**
  An attention is an externally generated signal to an executing task; the result is to "get the attention" of the task by interrupting its course of execution. Attentions include strong and weak interrupts signaled from the keyboard.

Exceptions

Both errors and attentions alter the flow of execution. In most cases, the executing function suspends and generates a diagnostic message. In the default case, the system displays the diagnostic message and requests "immediate execution" input. For certain errors and attentions, however, you can specify an alternate course of action. Errors and attentions within this class are called trapped errors and trapped attentions and are said to signal exceptions. The special procedures the system executes when it signals exceptions are called handlers.

An exception is a condition that can occur unexpectedly and to which an APL function can make a direct response. The initiation of the response to an exception is automatic and independent of the state of program execution. Whenever an exception occurs, execution suspends and the system automatically executes an exception latent expression, ⎕*ELX*.

System Response to Errors

The system response to errors is different for errors that occur during system commands, input, and function execution. System command errors are not execution errors and do not signal exceptions. Input errors and errors that result from evaluating quad (⎕) input, cause the system to display a diagnostic message and request input again. This behavior allows you to correct the error. Execution errors cause the system to suspend execution at the point of the error.

The system performs this procedure in response to an execution error:

1. The statement being executed is stopped, the system takes "clean up" actions, and discards temporary storage.

2. If the error occurs within a defined function, the system suspends execution on the line where the error occurred. The system line counter (⎕*LC*) and state indicator (⎕*SI*) reflect the point of suspension.

3. The system generates and records in ⎕*DM* a diagnostic message, if the error is a trapped error. If the error is not trapped, the system displays the diagnostic message.

4. If the error is trapped, the system executes ⎕*ELX* (⍎⎕*ELX*). If ⎕*ELX* holds its default value, '⎕*DM*', the effect is the same as it is for errors that are not trapped; that is, the system displays the diagnostic message.

5. If ⎕*ELX* does not call ⎕*ERROR* or restart execution by branching, an error stop occurs and the the function takes the stop action that you specified with ⎕*SA*.

6. The task requests immediate execution input.

An error stop occurs when execution stops because an error or attention was not trapped, or a handler failed to reinitiate execution with a branch or ⎕*ERROR*.

System Response to Attentions

A similar procedure takes place for attentions. An attention exception is signaled if a weak interrupt results from a signal. A weak interrupt occurs when you press Ctrl-Break.

An attention exception occurs when the next function line is reached after you signal a weak interrupt. The system then takes the following actions:

1. It suspends execution at the beginning of the next function line. The system line counter (⎕*LC*) and state indicator (⎕*SI*) reflect the next line to be executed.

2. The system generates and internally stores a diagnostic message, consisting of the function name and line number.

3. The system executes the statement ⎕*ALX* (⍎⎕*ALX*). If ⎕*ALX* holds its default value, the system displays the diagnostic message generated in Step 2.

4. If ⎕*ALX* did not restart execution, an error stop occurs. The system performs the stop action specified by ⎕*SA*, and the task requests immediate execution input.

Strong interrupts are not trapped; they do not signal an attention exception. If you press Ctrl-Break twice in quick succession, a strong interrupt is generated. When a strong interrupt occurs, the system takes the following actions:

1. If the interrupt occurs within a defined function, it suspends execution at the start of the line where it signaled the interrupt. The system line counter (⎕*LC*) and state indicator (⎕*SI*) reflect the point of suspension.

2. The system generates a diagnostic message and stores it internally. The message consists of the word *INTERRUPT* and an indication of the point of interruption.

3. The system displays the diagnostic message.

4. An error stop occurs. The system performs the stop action specified by ⎕*SA*, and the task requests immediate execution input.

Diagnostic Messages

When an error exception or attention exception occurs, the diagnostic message is recorded in the system variable ⎕*DM*. Therefore, ⎕*DM* contains the most recent diagnostic message recorded in the workspace. The diagnostic message is a character vector, often containing newline (⎕*TCNL*) characters. The value of ⎕*DM* indicates the nature of the exception and where it occurred. You can write error-handling routines to analyze the diagnostic message, then take appropriate action.

The form of ⎕*DM* is:

- Following an attention exception, ⎕*DM* contains a function name and line number and no newline characters. The function name and line number correspond to the top entry in the state indicator.

```
      ρ⎕DM ◊ ⎕DM
9
SAMPLE[4]
```

- Following an error exception generated by an error in immediate execution mode, ⎕*DM* contains the type of error and an indication of where the error occurred.

```
      ρ⎕DM ◊ ⎕DM
44
LENGTH ERROR
      MAT←A,[1] B
           ^
```

- Following an error exception generated by an error in an unlocked function, ⎕*DM* contains the type of error, the function name and line number, and an indication of where the error occurred.

```
      ρ⎕DM ◊ ⎕DM
52
LENGTH ERROR
SAMPLE[5]  MAT←A,[1] B
                ^
```

- Following a strong interrupt, *⎕DM* contains the word *INTERRUPT*, followed by the function name and line number If the system interrupts execution in the middle of a function line, the point of interruption is also indicated.

```
      ρ⎕DM ◊ ⎕DM
57
INTERRUPT
SAMPLE[5] ◊ INPUT←10↑⎕ ◊◊
                        ^
```

- Following either an error exception or another interruption of execution in a locked function, *⎕DM* contains an indication of the reason for halting and the function name and line number followed by a del-tilde (⍫).

```
      ρ⎕DM ◊ ⎕DM
20
WS FULL
SAMPLE2[9] ⍫
```

- In a clear workspace, or in a workspace where no diagnostic message is recorded, *⎕DM* is empty.

```
      ρ⎕DM ◊ ⎕DM
0
```

- Following the execution of *⎕ERROR*, *⎕DM* contains the argument that was supplied to *⎕ERROR*, followed by an indication of where the exception was signaled.

```
      ρ⎕DM ◊ ⎕DM
138
YOU DO NOT HAVE ACCESS TO THIS FILE;
CONTACT THE APPLICATION STEWARD.
SAMPLE[5] ◊ A←FREAD TIENO,COMP
                  ^
```

  If the workspace does not have enough storage to record the diagnostic message, the system displays *NO SPACE FOR ⎕DM*, followed by the diagnostic message that would have been recorded After *NO SPACE FOR ⎕DM* occurs, *⎕DM* is empty.

Signaling Intentional Exceptions

The system function *⎕ERROR* signals an error exception in the environment of the calling function (or in the global environment, if it is executed with an empty state indicator). The character vector argument to *⎕ERROR* is used by the system as the first line of the diagnostic message recorded when the exception is signaled. Useful applications of *⎕ERROR* include the following.

- **Defining New Errors**
  Using *⎕ERROR*, a program can generate an application error message that is functionally identical to an error message that occurs in execution. The exception that results invokes the execution of *⎕ELX*, making it possible for you to write a general error-handling routine that reacts to both APL errors and errors that you define.

- **Writing Utility Functions That Behave Like System Functions**
  Since *⎕ERROR* signals an error exception in the calling environment, you can write utilities that signal *DOMAIN ERROR*, *RANK ERROR*, *LENGTH ERROR*, and so on, at the point where the utilities are called with an erroneous argument.

Stop Action

Each time processing stops for immediate execution input, the system takes the **stop action** specified by the system variable *⎕SA*.

The system variable *⎕SA* can specify one of several possible actions you can take when a task stops for immediate execution input. The possible actions are:

- take no action (the default) — `⎕SA←''`
- exit from suspended and pendent functions until a "safe" environment is reached — `⎕SA←'EXIT'`
- signal a *STOP ACTION* if immediate execution mode is entered — `⎕SA←'ERROR'`
- clear the active workspace — `⎕SA←'CLEAR'`
- end the session by logging off — `⎕SA←'OFF'`.

Normally, ⎕*SA* is used as a "last resort" action, taken only if localized exception handlers are unable to prevent a program halt. If a user signals a strong interrupt to stop an application, you can use ⎕*SA* to exit from suspended or pendent functions, or to clear the workspace before you return control to the user or resume execution of the application.

Interactions between Errors and Attentions

If an error occurs because the value of ⎕*ELX* is an unexecutable APL statement, no exception is signaled (that is, ⎕*ELX* is not invoked again). If, however, a trapped error occurs within a function called when ⎕*ELX* is executed, an exception is signaled in that function. If an error occurs because the value of ⎕*ALX* represents an unexecutable APL statement, a trapped error occurs and ⎕*ELX* is invoked.

If an error occurs after you signal attention once but before the next function line is reached, the system gives priority to the error.

Basic Algorithms for Error Handling

The possible applications for error handling are almost limitless. This section examines a few of the fundamental algorithms.

Resignaling Errors in the Calling Environment

Using exception handling, you can make APL functions behave like system primitives. If an error occurs within an exception handling function, the system does not suspend execution at the point of error. Instead, it removes the function from the execution stack and signals the same error in the environment from which it called the function. If a similar handler is defined in the calling environment, the error can be resignaled in its calling environment, and so on. This technique is useful because it can pass an error "back through the stack" until a point is reached where the error can be handled differently or execution can be easily restarted.

The functions *ONE*, *TWO*, and *THREE* interact in this manner.

```
      ∇ ONE;⎕ELX
[1]     ⎕ELX←'⎕DM'
[2]     TWO
      ∇

      ∇ TWO;⎕ELX
[1]     ⎕ELX←'⎕ERROR(∧\⎕DM≠⎕TCNL)/⎕DM'
[2]     THREE
      ∇
```

```
     ∇ THREE;⎕ELX
[1]    ⎕ELX←'⎕ERROR(∧\⎕DM≠⎕TCNL)/⎕DM'
[2]    1÷0 ⍝ INTENTIONAL <DOMAIN ERROR>
     ∇
```

Notice the error handler defined in *TWO* and *THREE*. It takes the first line of the diagnostic message (which contains the type of error) and uses it as the argument to ⎕*ERROR*. An error is built into *THREE* to show the effect of ⎕*ELX*:

```
      THREE
DOMAIN ERROR
      THREE
          ^
```

Note that the error message shows the error to be external to the environment of *THREE*. At this point, the state indicator is empty.

```
      ⍴⎕SI
0 0
```

If you invoke the function *TWO*, which then calls *THREE*, the error is passed two levels back through the execution stack.

```
      TWO
DOMAIN ERROR
      TWO
        ^
```

Again, the state indicator is empty.

```
      ⍴⎕SI
0 0
```

In the function *ONE*, however, ⎕*ELX* has its default value. An error that occurs in *THREE* is passed back through the stack and becomes an error in *ONE*.

```
      ONE
DOMAIN ERROR
ONE[2] TWO
         ^
```

The state indicator shows execution suspended within *ONE*; ⎕*DM* reflects this specific error. ⎕*DM* reflects the most recent error in the workspace. In this case, three errors are signaled — once by the APL system and then twice by error handlers executing *⎕ERROR*.

```
      ⎕SI
ONE[2] *
      ⎕DM
DOMAIN ERROR
ONE[2] TWO
         ^
```

Error Logging

You can use exception handling to record errors that occur when an application is used. A record of these variables is very valuable, and it makes no difference how they occur — flaws in the application or user mistakes. For example, you could define a global error handler as

```
      ⎕ELX←'BUGREPORT ⋄ ⎕DM'
```

where the function *BUGREPORT* takes some action to the error (one approach is to append *⎕DM* to an "error file"). To the user, the error appears to have its normal effect.

Supplementing the Diagnostic Message

The standard APL error messages may not be helpful to a user who is not familiar with APL. You can define an error handler that displays instructions.

```
      ⎕ELX←'ERRORINSTRUCTIONS'
      ⍎⎕ELX
AN ERROR HAS OCCURRED IN THE PROGRAM THAT
YOU ARE USING.  PLEASE ENTER THE FOLLOWING
COMMAND:

      )SAVE 99 ERRORWS

AND THEN REPORT THE ERROR.
```

If your users prefer a language other than English, you can enhance the standard APL error reports to include the preferred language.

```
      WS FULL (VOTRE ESPACE DE TRAVAIL EST
PLEIN)
FOU[2] R←A,B
       ∧
```

Trapping Specific Errors

When a function executes ⎕*ELX*, ⎕*DM* contains the diagnostic message for the error that caused the function to execute ⎕*ELX*. Therefore, an error handler can examine ⎕*DM* to determine what kind of error occurred. This technique allows you to trap specific exceptions. (See the next section for another technique.)

For example, the function *RFAPPEND* in the next example uses exception handling to handle a *FILE FULL* error. This error occurs if you try to write a value to a file whose reserved space cannot hold the value. You must increase the file size before you can store the value. *RFAPPEND* uses ⎕*ELX* to examine ⎕*DM* and increases the file size if the error is *FILE FULL*. The default treatment of suspending and displaying the diagnostic message is given to other errors.

```
    ∇ VALUE RFAPPEND TIENO;OLDELX;⎕ELX
      ⍝  RESIZES FILE ON FILE FULL
       ⎕ELX←'→(''FILE FULL''≡9↑⎕DM)/RESIZE ⋄ ⎕DM'
      APPEND:0 0 ⍴VALUE ⎕FAPPEND TIENO
       →0
      RESIZE:⎕ELX←'⎕DM'
       (1E6+(⎕FSIZE TIENO)[3+⎕IO]) ⎕FRESIZE TIENO
       →APPEND
    ∇
```

Using the *HANDLERS* Workspace

The *HANDLERS* workspace contains functions you can use to simplify exception handling. This section provides an overview of the workspace, and describes how you use the three main functions.

The functions in the workspace work with ⎕*ALX*, ⎕*ELX*, and special labels or expressions, called handlers, which specify the action you want your function to take. Handlers are guided by public comments (⍝∇) and return executable expressions that are processed with format (⍎).

To use the handler facility, you perform the following actions.

- Reference the handler functions in ⎕*ALX* and ⎕*ELX*. When an exception occurs, the referenced function examines the handlers and takes the specified action.
- Build the proper labels or handlers into your functions. The handlers specify the attentions and errors you want to handle and the action you want to take when they occur.

Before you incorporate the *HANDLERS* facility into your application, decide how you want your application to handle errors and attentions. You can then determine which functions in the *HANDLERS* workspace you will need to copy into your application workspaces.

The *HANDLERS* workspace contains three main functions: *ALXHANDLER*, *ELXHANDLER*, and *HANDLERFOR*.

- *ALXHANDLER* handles attentions.
- *ELXHANDLER* handles errors such as *SYNTAX ERROR*, *FILE FULL*, and so on.
- *HANDLERFOR* handles all exceptions.

The auxiliary functions *ATTENTION* and *INTERRUPT* signal *ATTENTION* and *INTERRUPT* exceptions through ⎕*ERROR*, and allow your function to execute ⎕*ELX* and perform the action you want to take.

Table 10-1 summarizes the behavior of the functions in the *HANDLERS* workspace.

Table 10-1. *HANDLERS* Workspace Functions

Function	Description
ALXHANDLER	Determines an appropriate action and execution level in the state indicator. The action is based on the existence of the local identifiers *ALXsetup* and *ALXbranch* at each level.
ATTENTION	Signals an *ATTENTION* error.
ELXHANDLER	Determines an appropriate action and execution level in the state indicator. The action is based on ⎕*DM*, the handlers in the functions called at each level, and on the handlers in the variable *ELXglobalhandler*.
HANDLERFOR	Determines an appropriate action and execution level in the state indicator. The action is based on the exception specified in its right argument, on handlers in the functions called at each level, and on handlers in the variable *ELXglobalhandler*.
INTERRUPT	Signals an *INTERRUPT* error.

Handling Attentions with *ALXHANDLER*

ALXHANDLER is a convenient way to handle attentions independently of exceptions. It handles attentions by redirecting execution to a particular line in the controlling function, or by continuing execution as if the attention did not happen. The controlling function is the most local function that has *ALXsetup* or *ALXbranch* defined.

ALXHANDLER determines when to ignore an attention and where to branch using the special labels *ALXsetup* and *ALXbranch*. (*ALXsetup* and *ALXbranch* do not necessarily have to be labels; they can be local functions or variables.)

You can assign an optional global variable *ALXmessage* that contains the text of a message you want displayed when an attention is handled. No message will display if the variable does not exist. This variable is in the *HANDLERS* workspace with the default message *ATTENTION RECEIVED*. You can copy, alter, or dispose of this variable as you want.

Rules for Using *ALXHANDLER*

ALXHANDLER follows these rules.

- If the controlling function has not reached the line labeled *ALXsetup*, execution continues.
- If the controlling function is past the line labeled *ALXsetup*, the function branches to the line labeled *ALXbranch*.
- *ALXHANDLER* references *ALXsetup* and *ALXbranch* only once each. This behavior can be significant if they are functions rather than labels or variables.

- *ALXHANDLER* should be called directly by ⎕*ALX* and ⎕*ELX*; it should not be called by a cover function.

ALXsetup and *ALXbranch* follow these rules:

- They can be labels, local variables, or local functions. (If local functions, they should not set the global value of ⎕*IO*.)
- If only *ALXbranch* is defined in the controlling function, any attention causes a branch.
- If only *ALXsetup* is defined in the controlling function, execution continues if "setup" is not complete; otherwise, execution branches to *ALXsetup*.
- If either *ALXsetup* or *ALXbranch* is localized, but undefined in the controlling function, then attention is signaled to the function that calls the controlling function.
- If neither *ALXsetup* nor *ALXbranch* is local anywhere in the function execution stack, then the attention is handled as if ⎕*ALX* had its default value of '⎕*DM*'.
- A value of ¯1 for *ALXbranch* indicates that any attention should be ignored.

Example

To use *ALXHANDLER*, follow these steps.

1. Reference *ALXHANDLER* in ⎕*ALX*.

```
        ⎕ALX←'... ⋄ ⍎ALXHANDLER'
```

   You can store other expressions in ⎕*ALX*, but ⍎*ALXHANDLER* must be last.

2. Use *ALXsetup* and *ALXbranch* in the key functions of your application. The following example shows a typical use.

```
     ∇ REPORTS
.
.
.
[10] ALXsetup:'Align paper'
[11]  SINK←⎕INKEY
[12]  REPORT1
[13]  REPORT2
[14]  REPORT3
[15]  →0
[16] ALXbranch:
.
.
.
     ∇
```

If attention is signaled before line `[10]` of reports, it is ignored as if it never happened.

If an attention is signaled at any time during execution of the subfunctions, then execution resumes at line `[16]` of reports.

Handling Error Exceptions

ELXHANDLER and *HANDLERFOR* are convenient ways to handle error exceptions, like *SYNTAX ERROR* and *FILE FULL*. They work with special expressions called handlers.

When an error occurs, *ELXHANDLER* examines ⎕*DM* to see what error occurred, compares it against any handler defined for that error, and returns an appropriate expression to be executed.

HANDLERFOR is a variation of *ELXHANDLER* that gives you finer control of error handling than *ELXHANDLER*. *HANDLERFOR* takes the text of the error as an argument and returns the handler for that error as a result. *ELXHANDLER* is equivalent to *HANDLERFOR* ⎕*DM*.

You define handlers by placing them in special comments of your functions or in the global variable *ELXglobalhandler*. Handlers can apply to some or all exceptions, and can refer to a single line of code or an entire function. They can apply to a single function, or to a function and all of its subroutines. A global handler can handle any errors not explicitly handled.

How to Construct a Handler

Following these steps to construct a handler.

1. Enclose each handler in curly braces ({ }).
2. Enter the exceptions (names of errors) that this handler should receive. If you have more than one, separate them with colons (:). The exception is a text string that is compared to the beginning of ⎕*DM* (prior to the word *ERROR*); for example, *FILE TIE*, *SYNTAX*, or *INSUFFICIENT HANDLES*. You can use a star (⋆) as a wildcard character at the end of an exception. For example, *FILE*⋆ could handle *FILE TIE ERROR*, *FILE NAME ERROR*, and so on.

3. Enter the action you want the system to take if any of these exceptions occur. The action can be any APL expression that is a valid argument to the execute operator. Be sure to separate the errors from the action with a colon.

4. You can define more than one handler on the same function line. The system ignores anything between the right brace of the first handler and the left brace of the next handler.

5. To extend the scope of a handler, place a down arrow (↓) before the exceptions. The down arrow indicates that the action is to be executed in the function where the exception occurred, rather than in the function where the handler is defined. Otherwise, *⎕ERROR* will return to the environment of the handler before the APL expression is executed.

6. Place the handler in a public comment in one or more of your application's key functions, or assign it to the global variable *ELXglobalhandler*.

Figure 10-1 shows an example of two handlers placed in a public comment.

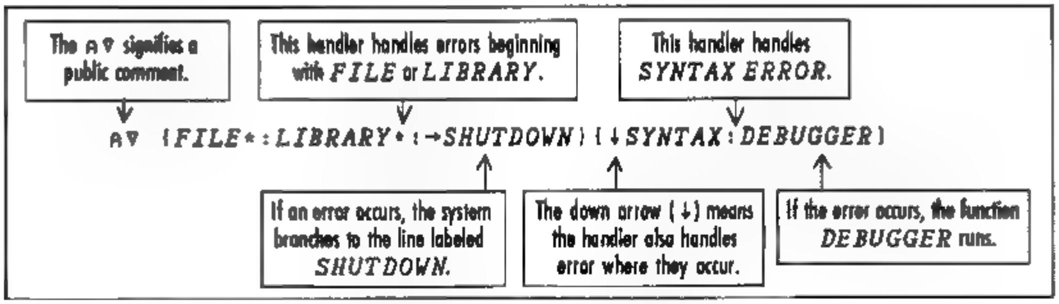

The ⍝∇ signifies a public comment.
This handler handles errors beginning with FILE or LIBRARY.
This handler handles SYNTAX ERROR.
⍝∇ {FILE*:LIBRARY*:→SHUTDOWN}{↓SYNTAX:DEBUGGER}
If an error occurs, the system branches to the line labeled SHUTDOWN.
The down arrow (↓) means the handler also handles error where they occur.
If the error occurs, the function DEBUGGER runs.

Figure 10-1. Handlers

Special Variables

Two special variables allow you to define global and function-wide handlers: *ELXglobalhandler* and *ELXfwhl*.

- You can define one or more handlers in the variable *ELXglobalhandler*. This variable can be global or local.
- You can assign a value to the variable *ELXfwhl*. This variable defines the line of the function that contains the handler definitions. In this way, you need not define all of the handlers on line 1 or on the lines where exceptions might occur. If you define *ELXfwhl*, be sure the definition is constant throughout your application.

Rules for Using *ELXHANDLER* and *HANDLERFOR*

- *ELXHANDLER* and *HANDLERFOR* will not work on functions that are shadowed by local objects in the ⎕*SI* stack.
- When an error occurs, the system inspects the public comment (if any) on the line where the error occurred to see if it contains a handler for the exception. If it does not contain a handler, the system inspects the public comment on the first line of the function.
- If line 1 does not contain a handler, the system inspects the public comment on the line that called the function. The system repeats this search until it finds a handler, or until it inspects all pendent functions associated with the most recent suspended function. If the system does not find a handler, it displays ⎕*DM*, and halts execution in the environment where the exception occurred.
- If *ELXglobalhandler* is global to all of the functions that the system is searching, and is not localized in any other function in the ⎕*SI* stack, then the system inspects its value only after it inspects all of the pendent functions.

- If *ELXglobalhandler* is local to one of the functions, the system inspects its value only after it inspects the handler line of that function.

Examples

The first example shows how you can set a "workspace-wide" global handler in *ELXglobalhandler* and specific handlers within an application function.

1. Copy *ELXHANDLER* into your application workspace.
2. Reference *ELXHANDLER* in ⎕*ELX*.
3. Define the default global handler for the workspace and assign it to the global variable *ELXglobalhandler*.

   ```
   ⎕ELX '... ⋄ ⍎ELXHANDLER'
   ```

   ```
   ELXglobalhandler←'{↓*:''Call Prog.''}'
   ```

   You can store other expressions in ⎕*ELX*, but ⍎*ELXHANDLER* must be last.

   The down arrow (↓) extends the scope of the workspace-wide handler in *ELXglobalhandler*.

4. Define handlers for specific errors and place them in public comments in your functions.

   ```
         ∇ UPDATE
   .
   .
   .
   [3]   L2:'EDNA' ⎕FTIE 1 ⍝∇ {FILE
         TIE:⎕FUNTIE 1⋄→L2}
   [4]   DIR←⎕FREAD 1 7
   [5]   DIR APPEND MAT ⍝∇{↓FILE FULL:RESIZE
         DIR[2]⋄→⎕LC}
   .
   .
   .
         ∇
   ```

If file tie error occurs on line `[3]`, the handler unties the conflicting file and tries again. If other errors occur, *ELXglobalhandler* is used.

If file full error occurs in append, the function resize increases the file reservation and execution continues.

The next example shows how you can use *HANDLERFOR* to handle attentions rather than *ALXHANDLER*. By using *HANDLERFOR*, you can use handlers to direct your function's behavior.

1. Copy *HANDLERFOR* into your application workspace.

2. Assign `'⍎HANDLERFOR ''ATTENTION'''` to `⎕ALX`.

3. Assign `'⍎HANDLERFOR ⎕DM'` to `⎕ELX`.

   ```
   ⎕ALX←'⍎HANDLERFOR  ''ATTENTION'''
   ⎕ELX←'⍎HANDLERFOR  ⎕DM'
   ```

   You can store other expressions in `⎕ALX`, but `⍎HANDLERFOR '...'` must be last. *HANDLERFOR* `⎕DM` is the same as *ELXHANDLER*.

4. Define the handlers for attentions and place them in public comments in your functions.

   ```
         ∇ REPORTS
   [1]    ⍝∇{↓ATTENTION:→⎕LC}
   .
   .
   .
   [9]    READY:'Align paper'⋄SINK←⎕INKEY
   [10]   R1:REPORT1 ⍝∇ {ATTENTION:→R2}
   [11]   R2:REPORT2 ⍝∇ {ATTENTION:→R3}
   [12]   R3:REPORT3 ⍝∇ {ATTENTION:→DONE}
   [13]   DONE:
   .
   .
   .
         ∇
   ```

   The down arrow (`↓`) extends the scope of the handler to the environment where the attention was signaled. If attention is signaled other than when lines `[10]`, `[11]`, or `[12]` have begun executing, even during the execution of a subfunction, then it is ignored as if it never happened. If an attention is signaled during execution of the functions *REPORT*1, *REPORT*2, *REPORT*3, or their subfunctions, then execution resumes at the next line of reports.

11
Advanced Techniques

This chapter describes programming techniques you can use to develop applications or to customize APL★PLUS II/386 to suit your particular needs. The chapter is organized as follows.

- "Debugging Tools" describes the system functions you can use to debug and optimize your functions.
- "Using Subwindows in Your Applications" explains how to use the window management tools to open and manipulate windows under program control.
- "Interfaces to the Non-APL Environment" contains techniques you can use to manage your computer's memory, access DOS from APL★PLUS II/386, and call assembler routines from APL★PLUS II/386.
- "Customizing Your System" explains four techniques you can use to tailor APL★PLUS II/386 to your particular needs:
  - — redefine your keyboard
  - — customize your system for a non-IBM environment
  - — change default settings with ⎕*POKE*
  - — build a custom Help file.

Debugging Tools

This section describes three useful system functions that you can use to debug and optimize the functions in your applications.

Function Trace and Stop

Two system functions, ⎕*TRACE* and ⎕*STOP*, can help you debug your programs by allowing you to see how the system executes each line of a function.

⎕*TRACE* turns on the tracing mechanism for a function. To set tracing for certain lines of a function, use ⎕*TRACE* dyadically. The left argument is an integer vector or singleton that indicates the line numbers you want the system to trace.

```
      (⍳5) ⎕TRACE 'MYFN'
```

To see which lines have been set, use ⎕*TRACE* monadically.

```
      ⎕TRACE 'MYFN'
```

The next time the function is run, ⎕*TRACE* follows the execution and displays the result of each line and branch. If lines were set previously, those line number are returned as the explicit result and the new trace settings are established. If no trace settings were previously set, the result is empty.

The following example shows a sample function and trace.

```
    ∇  RESULT←GO
[1]   'LINE OF OUTPUT!'
[2]   RESULT ← 1
[3]  LABEL: RESULT ← (0,RESULT)+RESULT,0
[4]   →LABEL × 4 > ⍴RESULT
    ∇

      (⍳5) ⎕TRACE 'GO'
      GO
GO[1] LINE OF OUTPUT!
GO[2] 1
GO[3] 1 1
◊   →3
GO[3] 1 2 1
◊   →3
GO[3] 1 3 3 1
◊   →0
1 3 3 1
```

The last line shows the explicit result of *GO*.

Trace settings are saved when you save the workspace, but they are not copied with the function. The editor shows which lines have trace settings. You can modify trace settings in the editor with Ctrl-,.

To remove all trace settings from a function, execute:

```
      ⍬ ⎕TRACE fnname
```

Although ⎕*TRACE* can be useful in showing you how a function executes, you may sometimes want to stop a function execution altogether. This technique would allow you to examine the local environment; for example, to see the values of local variables created by the function. ⎕*STOP* does exactly that.

To stop the function at certain lines, specify those lines in the left argument as an integer vector or singleton. If you set any stops previously, those line numbers are returned as the explicit result and the new stop settings are established. If no stop settings were previously set, the result is empty.

```
      (⍳3) ⎕STOP 'MYFN'
```

The next time you run the function, APL stops and enter immediate execution mode before executing any line that has a stop setting.

Use ⎕*STOP* monadically to see the stop settings in a function. The right argument is the name of the function you want to halt.

```
      ⎕STOP 'MYFN'
1 2 3
```

Stop settings are also saved with the workspace, but are not copied with the function. The editor shows which lines have stop settings. You can modify stop settings in the editor with Ctrl-Period (.).

To remove all stop settings, execute:

```
        ⍬ ⎕STOP fnname
```

Function Timing and Monitoring

The ambivalent system function ⎕*MF* and the functions in the *MFFNS* workspace provide a mechanism for exploring where the time is used in a function or group of functions. This allows efficient use of your time in optimizing the code.

Timing Function Lines

The technique for timing the individual lines of the functions being studied follows.

1. Use 1 ⎕*MF* '*fnlist*' to begin the line-by-line monitoring of where the time is spent.

2. Run the programs. The system will accumulate the monitoring information.

3. Use ⎕*MF* '*fnname*' to capture the timing data accumulated for each function monitored for study, possibly with the report functions in the supplied workspace *MFFNS*.

4. Use 0 ⎕*MF* '*fnlist*' to end the timing. Timing itself takes time, slowing the functions; and storing results takes space, increasing function size.

Note: The time units returned by ⎕*MF* can be arbitrary and varies among the different APL★PLUS Systems. Check the actual result of *MFRESOLUTION* in the *MFFNS* workspace for the duration of the unit in seconds.

Using the High-Resolution Timer

The APL★PLUS II/386 default timer is based on the DOS clock resolution of 18.2 ticks per second. In the infrequent cases when greater resolution of timings is needed, the default DOS timer can be replaced with a more precise user-supplied timer (a terminate-and-stay-resident routine). In the case of IBM or fully compatible machines, a high resolution timer, RESTIME.COM, is provided with the system. This handler gives 214.552 microsecond resolution time values. (For other hardware, you would have to supply the timer routine.)

To use RESTIME.COM in place of the built-in DOS timer mechanism, follow these steps.

1. **Before** you start the APL★PLUS System, load the timer interrupt handler (user-supplied or RESTIME.COM). **Do not load the timer from ⎕*CMD*.**
2. Start your APL★PLUS System.
3. Tell the system which timer to use, either with the function *MFHIRES* from the *MFFNS* workspace or, for a user-supplied timer, by using *intnum* ⎕*POKE* 109 to put the interrupt number into memory location 109.
4. Remove the timer interrupt handler with RESTIME /R when you leave APL.

The *MFHIRES* function allows you to switch to and from the use of a separate resident high resolution timer routine as an alternative to the default DOS clock.

The *MFRESOLUTION* function in the *MFFNS* workspace reports the duration in seconds of one unit of the timer currently in use; for example:

```
      ÷ MFRESOLUTION   Show time units per second
18.2                   Standard resolution
```

MFRESOLUTION is required by each of the main timing functions from that workspace as a subroutine.

```
      MFHIRES 1        Use high-resolution timer
0                      Was low resolution

      ÷ MFRESOLUTION   Show current ticks per second
4660.874753            High resolution timer in use
```

Running *MFHIRES* 0 restores the default timing.

If no high-resolution timer has been loaded, the *MFHIRES* function says so, and leaves default timing in effect.

Warning: If you fail to load a timer, *⎕MF* results are undefined and can crash APL.

If you supply your own high-resolution timer, you must modify the unlocked functions *MFRESOLUTION* and *MFHIRES*.

Note for assembly language programmers: While essentially useless for APL code, the timer RESTIME.COM has an ultra-high resolution for use in timing assembler code being developed for use with *⎕CALL*. This timer has a time unit = 0.838 microseconds. The timer resolution is activated by placing in AX a 0 for high resolution, a 1 for ultra-high resolution, or a 2 to reinitialize, before calling interrupt E0H (224). The number of ticks is reported as a four-byte integer in registers DX and AX for the high-resolution timer, or the six-byte integer in registers AX, BX, and CX for the ultra high-resolution timer.

Using the *MFFNS* and *TIMER* Workspaces

Besides *MFHIRES* and *MFRESOLUTION*, the main functions in the *MFFNS* workspace are ⎕*MF* reporting functions for timing APL code:

- *CUMSUM*
  Profiles an entire workspace or application within it. Generally used first to identify which functions in the application use the most time.
- *LTIMES*
  Profiles the individual lines of a function. Shows line-by-line iterations and execution time within functions. Use it on the bottleneck functions identified by *CUMSUM*.
- *EXT*
  Times execution of an individual APL expression.

These functions are described in more detail in "Programming Tools" in the *Utilities Manual*. To use them, copy them into the workspace being timed, after checking for name conflicts. After your analysis, you can erase all of the functions and variables copied from the *MFFNS* workspace by executing:

```
      ⎕ERASE GRPMFFNS
```

A small statistical application for demonstrating the use of ⎕*MF* and the *MFFNS* workspace functions is provided in the *TIMER* workspace. See the *DESCRIBE* in that workspace for details.

Using Subwindows in Your Applications

This section explains how to use the APL★PLUS II/386 window management facilities to create and manage windows under program control. It explains how you open and close application windows and edit windows, and control window settings. This section also summarizes the *SUBWNDWS* workspace, which contains examples that show you the capabilities of these features.

Note: For information on how to use subwindows in the session manager, see the "Using the Session Manager" chapter.

What Is a Window?

A window is a rectangular area of the screen. You can use windows to interact with APL, edit objects, or run programs. APL★PLUS II/386 has three types of windows: the APL window, edit windows, and application windows.

The APL window is where the APL session and terminal mode input and default output occurs.

Edit windows are where you edit APL objects. From an edit window, you can create new edit windows or move through the session manager edit ring. In an edit window, APL★PLUS II/386 formats and scrolls the data in the window, controls the way you interact with the data, and handles the different ways you can exit.

Application windows are where programs can run secure and separate from the APL session or edit sessions. You must supply and format the data for an application window, define the user interaction, and establish the exit rules. You can use *⎕WGET* and *⎕WPUT* to retrieve and write screen images, and you can use ⎕*WIN* to gather input in discrete fields within the window. Unless you are using ⎕*WIN* to gather input, input is prohibited in an application window.

Every window in the session manager has a window handle assigned to it. The window handle is a non-negative number. The APL session always has window handle 0; other windows have positive numbers. You use this window handle with other window functions to identify the window. (This is similar to using a file tie number to identify a file.) The system assigns the handles as the windows are created and does not reuse handles after windows are closed.

As a programmer, you generally use the APL and edit windows while you are developing an application. You use application windows to actually run the application and gather input from users.

Opening and Closing Application Windows

To open a new application window, use the system function ⎕*WOPEN*. ⎕*WOPEN* creates the window according to your specifications and by default places it in front of other windows in the session manager edit ring. The syntax is:

```
    whdl ← ⎕WOPEN wparms
```

The argument is a heterogeneous vector of window parameters that specify the characteristics of the window:

- The first element is the name of the window. This element cannot be empty.
- The second through fifth elements specify the upper-left corner and size of the window: starting row, starting column, number of rows, and number of columns. You must either specify all of the elements or omit all of these elements. The default is `0 0,⎕CRT[5 6]`.

- The sixth element specifies the depth in the display. By default, this element is 0, which specifies the front session. A non-zero value indicates the number of sessions to be placed in front of this session. This value is reduced if it exceeds the number of sessions available to be placed in front.

- The seventh element determines if the window is visible or invisible. The default is 1, visible; 0 specifies invisible.

- The eighth element specifies the window type; currently, this must be *'AD'* (the default), for application data window.

⎕WOPEN creates the window and assigns it the next available window handle. The handle is the result of *⎕WOPEN*.

The following example creates an invisible application window and assigns it to window handle 1. The upper-left corner is the upper-left corner of the screen, and the window has 10 rows and 30 columns.

```
      ⎕WOPEN 'TEST' 0 0 10 30 1 0 'AD'
1
```

You do not see the window, because the seventh element specifies that the window is invisible; however, the window exists with the window handle 1.

To make the window visible, use *⎕WCTRL*. (See the "*⎕WCTRL*" section in the *Reference Manual*.) To close an open window, use the function *⎕WCLOSE*. *⎕WCLOSE* deactivates an application or edit window and removes it from the session manager ring.

Caution: *⎕WCLOSE* closes a window even if the session in the window contains changed data. Because *⎕WCLOSE* does not prompt you to save any changes you may have made, your application must first extract any data you need to save.

The syntax of *⎕WCLOSE* is:

⎕WCLOSE whdls

The right argument contains the window handles of the windows you want to close.

To determine the window handles of the open windows in the session manager ring, use the function ⎕*WNUMS*. ⎕*WNUMS* returns the handles of all open windows. Remember: the APL session always has handle 0. Attempts to close the APL session are ignored.

To close all of the open windows in the ring, use ⎕*WNUMS* as the argument to ⎕*WCLOSE*; for example:

```
        ⎕WCLOSE ⎕WNUMS
```

You can use ⎕*WCTRL* to change the order of the sessions in the ring. To move a session, place the number of sessions in front of it into the sixth element of the left argument to ⎕*WCTRL*.

To bring the APL window to the front, press Ctrl-Esc.

Controlling Window Settings

Three functions allow you to retrieve or change window parameters and state settings: ⎕*WCTRL*, ⎕*ECTRL*, and ⎕*WSTATE*.

Setting and Changing Window Parameters

⎕*WCTRL* retrieves and changes the window parameters for application and edit windows. These parameters are the values used with ⎕*WOPEN* to set window characteristics. To retrieve window parameters for one or more windows, use the syntax:

```
    wparms ← ⎕WCTRL whdls
```

The right argument is one or more window handles. If you specify a scalar handle, the result is a vector containing the windows parameters for the window with that handle. If you specify a vector of handles, the result is a matrix where each row contains the window parameters for the corresponding window handle. The eight columns contain the following information:

- The first column is the name of the window.
- The second through fifth columns specify the upper-left corner and size of the window: starting row, starting column, number of rows, and number of columns.
- The sixth column specifies the depth in the ring; 0 means the front.
- The seventh column determines if the window is visible (1) or invisible (0).
- The eighth column specifies the window type: APL, AD (Application), CV (Character vector), CM (Character matrix), FN (Function), or NM (Numeric matrix). (For information on the types of windows, see the "Sessions and Windows" section in the Editing with the Session Manager chapter in this manual.)

To change the parameters for a single window, use the syntax:

```
    wparms ⎕WCTRL whdl
```

The right argument is the window handle of the window you want to change. The left argument is a heterogeneous vector that contains the changes:

- The first element is the name of the window. You can specify an empty element as a placeholder if you want to change the other elements without specifying a new name.
- The second through fifth elements specify the upper-left corner and size of the window: starting row, starting column, number of rows, and number of columns. You must either specify or omit all of these elements. The default is `0 0,⎕CRT[5 6]`.
- The sixth element specifies the depth in the ring, and contains the number of sessions you want placed in front of the session. If it exceeds the maximum available depth, it will be reduced. Currently, this element must be 0 (the default), which specifies the front.

- The seventh element determines if the window is visible or invisible. The default is 1, visible.
- The eighth element specifies the window type. This element is optional when you set parameters. You cannot change a window to types AD (Application), APL, or NM (Numeric matrix).

You only need to supply the minimum number of elements to effect the change. You can substitute an empty vector in place of the current name.

To display a representation of the contents of the edit ring, enter:

```
        ⎕WNUMS,⎕WCTRL ⎕WNUMS
```

Setting and Changing Edit Parameters

⎕ECTRL retrieves and changes the window edit related parameters for an edit window or the APL window. To retrieve edit parameters for one or more windows, use the syntax:

```
    eparms ← ⎕ECTRL whdls
```

The right argument is one or more window handles. If you specify one handle, the result is a vector containing the windows parameters for the window with that handle. If you specify more than one handle, the result is a matrix where each row contains the edit parameters for the corresponding window handle.

The result, *eparms*, contains the following elements:

- The first element determines whether the edit session is in browse mode (0) or edit mode (1). The default is 1.
- The second element controls the boxing style of the window. A 0 means the window is not boxed; a 1 means the window is boxed. The default is the value of *⎕WSTATE*[29].

- The third element controls the footer style. A 0 means no footer, a 1 means the standard status line, and a 2 means a centered window name. The default is the value of `⎕WSTATE[30]`.

- The fourth element controls the line numbering style for character editing sessions. A 0 turns line numbers off; a 1 turns line numbers on. The default is the value of the `editnums` startup parameter.

- The fifth element determines if horizontal scrolling is in effect. A 0 turns horizontal scrolling off; a 1 turns horizontal scrolling on. The default is the value of `⎕WSTATE[31]`.

- The next five elements control the window, footer and tagging color attributes. The sixth element is the inner window color; the default is `1↑editattrs`. The seventh element is the footer color; the default is `1↑1↓editattrs`. The eighth and ninth elements control the partial and full tag colors; the defaults are `1↑tagattrs` and `1↑1↓tagattrs`. The 10th element is the wrap attribute color; the default is the value of `wrapattr`.

- The 11th element determines if the window can be moved or resized. A 0 means the window is fixed; that is, you cannot move or resize it. A 1 means you can move the window but you cannot resize it. A 2 means you can resize the window, but you cannot move the upper-left corner from its anchor position. A 3 means you can move and resize the window. The default is 3.

- The next two elements (the 12th and 13th) specify the upper-left corner of the data shown in the window by row, then column. The default is 0 0.

- The next two elements (the 14th and 15th) specify the cursor position by row and column. This position is relative to the inner window; the default is 0 0.

- The 16th element specifies the tag state. The default is 0, no tag. A 1 means a tagged line or text, and a 2 means a tagged rectangle.

- The next four elements specify a tagged area, if any, by starting row, starting column, ending row, and ending column. The default is ¯1 ¯1 ¯1 ¯1.
- The 21st element is reserved.
- The 22nd element specifies the exit keystroke that left the window and returned to the APL session.
- The 23rd element indicates whether the data in the window changed. The default is 0, unchanged.
- The 24th element indicates whether the window was moved or resized. A 0, the default, means the window was not changed. A 1 means the window was moved. A 2 means the window was resized. A 3 means the window was moved and resized.

To change edit parameters, use the syntax:

```
    eparms ⎕ECTRL whdl
```

The right argument is the handle of the window whose edit parameters you want to change. The left argument is a vector containing the new edit parameters, as described above. You only need to specify the minimum number of parameters necessary to effect the change you want.

⎕*ECTRL* only works on edit windows or the APL window. If you try to use ⎕*ECTRL* on an application window, you receive an error message.

Determining and Changing Window Settings

⎕WSTATE defines the overall settings for all windows. These settings specify compatibility levels, interleaving behavior, output destinations, and window display styles. To display the current settings, use the syntax:

```
    state ← ⎕WSTATE
```

The result is a 31-element vector of state settings.

- The first four elements list the window manager version number (default=1), the subwindowing compatibility level (1=allow these features), the *⎕ED* compatibility level (1=APL★PLUS II/386 Version 4, 0=Version 3), and the maximum number of sessions available (constant 32).
- The fifth element determines how the APL session behaves with respect to other sessions. There are two aspects: (1) whether the APL session owns the entire screen, or only the area inside *⎕WINDOW*, and (2) whether the APL session participates in window rotation or remains at the back of the stack of windows while other windows rotate in front of it.
  - ❑ 0
    The APL session is unprivileged in size; that is, it only owns the region defined by *⎕WINDOW*. The APL session acts like every other session in the ring.
  - ❑ 1
    The APL session occupies the entire screen. When the APL session is active, it blocks all other windows regardless of the value of *⎕WINDOW*.
  - ❑ 2
    The APL session is unprivileged in size; that is, it only owns the region defined by *⎕WINDOW*. All subwindows appear in front of the APL session when the session is inactive. The APL session can obscure windows when it is active, but not when it is inactive.

- 3
  The APL session occupies the entire screen. All subwindows appear in front of the APL session when the APL session is inactive. This is the default.

- The sixth, seventh, and eighth elements define various output windows. The sixth element specifies the window for any undirected ⎕WGET or ⎕WPUT actions, and for ⎕WIN. The seventh element specifies the window that will receive output generated by 6 ⎕ARBIN, 7 ⎕ARBIN, and 9 ⎕ARBIN. The eighth element specifies the window for quad and quote-quad output. The default for all of the elements is 0, the APL session.

- The ninth and 10th elements specify the display attributes for the background and fill character for the screen region that is outside of any defined window. It exists only if all windows are less than full size and if the interleaving behavior (element 5) is 0 or 2. Specify the background as a display attribute number; specify the fill character as its origin 0 ⎕AV index. The defaults are 7 (background) and 32 (space fill character).

- The next 18 elements (11 through 28) specify the characters that form the optional box and icons surrounding the window. These are as follows.

  - 11 Upper-left corner; default = 213 (╒)
  - 12 Top line; default = 205 (═)
  - 13 Upper-right corner; default = 184 (╕)
  - 14 Left-side line; default = 179 (│)
  - 15 Right-side line; default = 179 (│)
  - 16 Lower-left corner; default = 192 (└)
  - 17 Bottom line; default = 196 (─)
  - 18 Lower-right corner; default = 217 (┘)
  - 19 Left scroll mark; default = 6 (←)
  - 20 Right scroll mark; default = 26 (→)
  - 21 Up scroll mark; default = 24 (↑)
  - 22 Down scroll mark; default = 25 (↓)
  - 23 Maximize token; default = 94 (^)
  - 24 Restore token; default = 4 (◊)
  - 25 Vertical thumb; default = 177 (▒)

- ❑ 26 Horizontal thumb; default = 177 (▓)
- ❑ 27 Upper-left menu token; default = 177 (▓)
- ❑ 28 Reserved

■ The 29th element specifies whether or not a new session is boxed. The default is 0.

- ❑ 0 A new session is not boxed.
- ❑ 1 A new session is boxed.
- ❑ 2 A new session inherits its boxing style from the parent session.

■ The 30th element specifies the footer style for a new session. The default is 1.

- ❑ 0 A new session does not have a footer.
- ❑ 1 A new session has a status line.
- ❑ 2 A new session has a footer consisting of the session name centered in the footer line.

■ The 31st element specifies whether horizontal scrolling is in effect for a new session. The default is 0.

- ❑ 0 A new session does not have horizontal scrolling.
- ❑ 1 A new session has horizontal scrolling.

You can reassign all of the elements of ⎕*WSTATE* by supplying the entire 31-element vector. You can reassign any element of ⎕*WSTATE* with indexed assignment. For example, to turn boxing on for new sessions, enter:

```
        ⎕WSTATE[29]←1
```

You can also use the `wstate` startup parameter to reassign ⎕*WSTATE*. See the Getting Starting chapter for details.

Using the Editor in Your Applications

In addition to creating application windows under program control, you can also create and use edit windows under program control.

Use the ⎕*ED* function to open and close edit windows. ⎕*ED* allows you to edit a character vector, character matrix, or numeric matrix.

The syntax of ⎕*ED* is:

newstate newval ← *state exitkeys title* ⎕*ED value*

The left argument is an integer vector or a nested vector with one, two, or three items. If the left argument is an integer vector, it defines the initial editor state. If the left argument is a nested vector, it can define the initial editor state, the keys that cause an exit, and a title for the editing session. The right argument is the character vector or matrix you want to edit.

⎕*ED* uses a special editing session in the session manager to edit the object you specify You can use all of the usual session manager editing operations, except that you cannot change the object's type and you cannot move around the ring. In addition, you cannot use the session manager menus under ⎕*ED*.

When ⎕*ED* ends, the system deletes the session from the edit ring, but you can override this. The edit subwindow optionally becomes part of the APL session image, so your application can restart ⎕*ED* without any screen flash. The system restores the cursor to the same position it had before ⎕*ED* was executed.

Defining the Initial Editor State

The *state* item defines the initial editor state. It can have a length of 0, 3, 4, 8, or 12 to 27 (the 13th element is only significant in the result). The *state* specifies:

- the location of the edit window
- the cursor position on entry to the editor
- the edit options, such as keyboard and pad states
- the size of the edit window
- a tagged region
- the colors of the edit session and status line
- the colors of the partially and fully tagged regions
- the color of the wrapped-line marker
- the handle of the session left behind or to re-enter
- the boxing style
- the footer style
- whether horizontal scrolling is on
- how you can modify the window
- the window modification style
- the window modification flag
- the data modification flag.

The first element specifies the line you want at the top of the screen. The next two elements define the initial cursor position by row and column.

The fourth element is the sum of the initial options. The allowable option codes are:

- 1 — line numbers on (always on in numeric sessions)

- 2 — insert mode on

- 4 — APL keyboard on

- 8 — NumLock on

- 16 — CapsLock on

- 32 — incorporate edit screen into APL session

- 64 — leave this session as a regular session in the ring when ⎕*ED* exists, assigning element 20
- 128 — re-enter a session whose handle is in element 20
- 256 — indicate whether the data in the session changed.

For example, to turn line numbers on and use the APL keyboard in a character session, use 5 as the fourth element of the left argument. The system remembers the original state of the keyboard and restores it when ⎕*ED* exits.

The next four elements define the editing subwindow by starting row, starting column, ending row, and ending column. Use ¯1 to specify current default dimensions; for example

```
      0 ¯1 15 ¯1
```

specifies a window 15 lines high that begins on line 0.

The next four elements define a tagged region. Tagged regions in character sessions do not have to be rectangular. Element [9] specifies the first tagged line; element [10] the first tagged column in the first tagged line. Elements [11] and [12] specify the last tagged line and last tagged column in the last tagged line. To specify no tagging, use ¯1 as element [9].

The system ignores the 13th element when it is used as input; the element is assigned in the result. The 14th element specifies the starting column number in numeric sessions, or whether horizontal scrolling is in effect during character sessions.

The next five elements specify color attributes.

- Element [15]
  Specifies the color attribute of the edit session.
- Element [16]
  Specifies the color attribute of the edit session status line.
- Element [17]
  Specifies the color attribute of a partially tagged region.

- Element [18]
  Specifies the color attribute of a fully tagged region.

- Element [19]
  Specifies the color attribute of the wrapped-line marker.

The next six elements specify editing and reporting features you can use when you enter and exit ⎕*ED*.

- Element [20]
  This element is the handle of the session left behind when feature 64 is used or the handle of the session you want to re-enter when you use feature 128. This element is 0 if no session is left. You can specify 0 to create a new edit session.

- Element [21]
  Specifies whether the session is boxed. A 0 means the window is not boxed; a 1 means the window is boxed. The default is the value of ⎕*WSTATE*[29]. This element is analogous to ⎕*ECTRL*[2].

- Element [22]
  Specifies the footer style. A 0 means no footer, a 1 means the standard status line, and a 2 means a centered window name. The default is the value of ⎕*WSTATE*[30]. This element is analogous to ⎕*ECTRL*[3].

- Element [23]
  Specifies whether horizontal scrolling is in effect. A 0 turns horizontal scrolling off; a 1 turns horizontal scrolling on. The default is the value of ⎕*WSTATE*[31]. This element is analogous to ⎕*ECTRL*[5].

- Element [24]
  Specifies how you can modify the window. A 0 means the window is fixed; that is, you cannot move or resize it. A 1 means you can move the window but you cannot resize it. A 2 means you can resize the window, but you cannot move the upper-left corner from its anchor position. A 3 means you can move and resize the window. The default is 3. This element is analogous to ⎕*ECTRL*[11].

- Element [25]
  Indicates whether the window was moved or resized. A 0, the default, means the window was not changed. A 1 means the window was moved. A 2 means the window was resized. A 3 means the window was moved and resized. This element is analogous to ⎕*ECTRL*[24].

- Element [26]
  Specifies the data modification flag; 0 means browse, 1 means edit. This element is analogous to ⎕*ECTRL*[1].

Note: Elements [1], [9], [10], [11], [12], and [14] are ⎕*IO* sensitive in numeric sessions. Element [27] is reserved for future use.

Defining the Exit Keys

The *exitkeys* item defines the keys that cause ⎕*ED* to exit. It is a character vector or integer vector of up to 32 integers. If you omit this item, only Ctrl E and Ctrl-Q cause an exit. Other keys or mouse movements that affect editing sessions in the session manager are prohibited, unless you explicitly define them as exit keys. The only effect of these keys is to exit ⎕*ED*; you must decide how you want your application to process them.

For example, you could define Ctrl-H as an exit key to have ⎕*ED* exit on a call to a Help facility. Your application would then display the help and resume execution of ⎕*ED*.

If you specify a mouse click, it triggers an exit only if you click the mouse outside the window occupied by ⎕*ED*.

Table 11-1 shows the exit keys and prohibited keys.

Table 11-1. Exit Keys and Prohibited Keys

Event	Key	Intended Action
		Default Exit Keys
276	Ctrl-E	Save and exit
415	Ctrl-Q	Quit and discard changes
		Default Prohibited Keys
262	Ctrl-A	Change rank/type
264	Ctrl-H	Display help panel
278	Ctrl-I	Initiate new session
308	--	Mouse click, left button
309	--	Mouse click, right button
395	Ctrl-V	Assign tagged lines to variable
398	Ctrl-X	Move to APL session
417	Ctrl-S	Save without exit
465	Alt-F6	Display system help
494	Ctrl-Shift-P	Move in edit ring
495	Ctrl-Shift-N	Move in edit ring
496	Ctrl-Shift-R	Rename session
498	Ctrl-Shift-D	Execute DOS command
499	Ctrl-Shift-I	Edit object at cursor
501	Ctrl-/	Show session manager menu

Defining the Title

The title is a character vector that contains the text you want placed on the status line. It starts in column 0. The default is no title.

Interpreting the Result

The result of ⎕*ED* is a two-item nested vector. The first item defines the exit from ⎕*ED*. Its data has the same meaning as *state*, and includes the event number of the exit key as element `[13]`. The second item contains the form of the object being edited, and reflects all changes made to it, even if the exit key was Ctrl-Q.

You use the information in the result to see how the user exited ⎕*ED*, to continue further processing, and perhaps to resume editing, possibly using *newstate* as the input state vector. You can see the changes made to the edited object.

For more information on ⎕*ED*, see the System Functions, Variables, and Constants chapter in the *Reference Manual*.

The *SUBWNDWS* Workspace

The supplied *SUBWNDWS* workspace contains example functions and utility functions to help you use the windowing and editing functions.

The *SETUP* and *SETUP0* functions create and display several different types of windows using a variety of colors and other options. You can run this function and examine the code to see how the function creates the windows.

The *WINTEST* function demonstrates the use of ⎕*WIN* in both application and edit windows. The comments in the function explain how the function creates, decorates, and uses the different types of windows as the focus for *⎕WIN*.

The *CASCADE* function illustrates how you can set up and place windows in a cascading pattern on the screen.

Other utility functions include functions that help you summarize the contents of the edit ring, display the names of windows, and close all open windows.

The *SUBWNDWS* workspace also contains *HELPEDIT*, a utility function to help you write custom Help files. See the "Building a Custom Help Facility" section.

Interfaces to the Non-APL Environment

This section describes:

- how you manage the memory in your 80386 computer
- how you can access non-APL environments from your APL session
- how you call assembler routines with the system function ⎕*CALL*. (For information on ⎕*NA* and using associated functions to call external routines, see the Using ⎕*NA* chapter in this manual.)

Memory Management

Version 3 or later of APL★PLUS II/386 differs from earlier versions in that the session manager is integrated into the APL interpreter and runs in extended memory. This integration greatly reduces competition between APL★PLUS II/386 and software that runs in the 640K DOS region.

Depending on your environment, APL★PLUS II/386 can use extended memory, Extended Memory Specification (XMS), and Expanded Memory Specification (EMS). If you do not use any type of memory management software, APL★PLUS II/386 runs in extended memory.

There are several software programs that help you manage the memory beyond the 640K DOS region. Version 5.0 of DOS provides two: HIMEM.SYS and EMM386.EXE. You can use these two programs together or you can use HIMEM.SYS by itself.

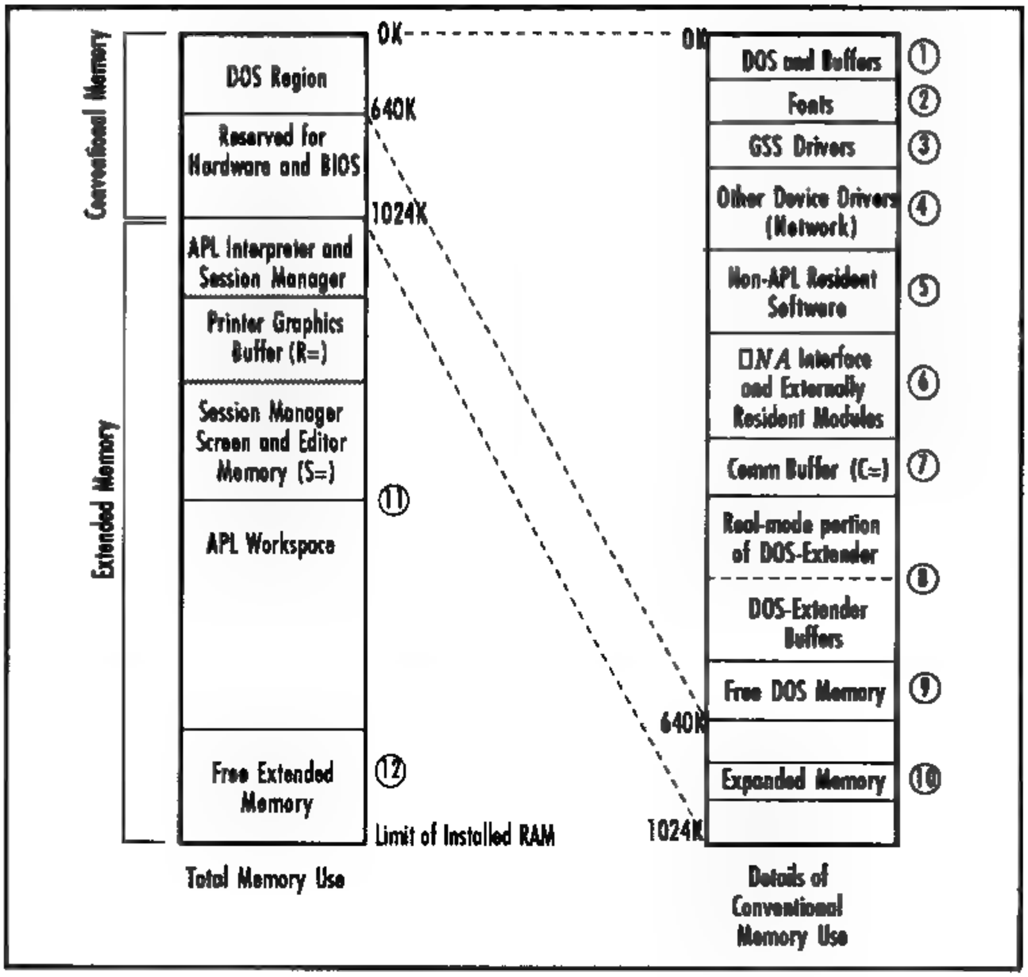
If you use HIMEM.SYS by itself, APL★PLUS II/386 uses all of the XMS memory that HIMEM.SYS provides. If you use

EMM386.EXE and HIMEM.SYS, APL★PLUS II/386 uses only the EMS memory that EMM386.EXE provides.

If you use a third party memory management program, such 386Max from Qualitas or QEMM from Quarterdeck, you should set them up so that they convert all memory to EMS. APL★PLUS II/386 uses only EMS memory when you use separate EMS and XMS drivers.

Memory Use

Figure 11-1 shows how a typical collection of programs and their data areas are laid out in the memory of an 80386 computer. This section describes the adjustments you can make and their implications. The circled numbers in the figure correspond to the notes that follow Figure 11-1 about managing DOS memory.



Conventional Memory
Extended Memory
DOS Region
Reserved for Hardware and BIOS
APL Interpreter and Session Manager
Printer Graphics Buffer (R=)
Session Manager Screen and Editor Memory (S=)
APL Workspace
Free Extended Memory
0K
640K
1024K
Limit of Installed RAM
⑪
⑫
Total Memory Use
0K
DOS and Buffers ①
Fonts ②
GSS Drivers ③
Other Device Drivers (Network) ④
Non-APL Resident Software ⑤
⎕NA Interface and Externally Resident Modules ⑥
Comm Buffer (C=) ⑦
Real-mode portion of DOS-Extender
DOS-Extender Buffers ⑧
Free DOS Memory ⑨
640K
Expanded Memory ⑩
1024K
Details of Conventional Memory Use

Figure 11-1. Memory Map of an 80386 Computer Running APL★PLUS II/386

1. DOS and its buffers — you can adjust the space used by DOS with the `BUFFERS=` and `FILES=` parameters in the CONFIG.SYS file. Smaller buffer space implies slower file I/O, even for APL file and workspace operations that are processed by the 386|DOS-Extender. (See Note 8 for more information on the DOS-Extender.)

2. Fonts — APL character fonts for the EGA, VGA, and some other displays, and the DOS Code Page facility allocate DOS space.

3. GSS drivers — these require at least 50 kilobytes of memory or more for high-resolution devices. If you do not always use the VDI graphics capabilities with APL★PLUS II/386, use the `/T` option in your CONFIG.SYS file or use a CGI.CFG file to load graphics only when required. See the *Installation Instructions* or the GSS Manual for more information.

4. Other device drivers such as local area network drivers can consume considerable amounts of space.

5. Many utility programs work as TSR routines, including DOS utilities (like FASTOPEN), third-party utilities (like Borland's Sidekick), and STSC utilities (like APLPRINT.COM).

6. Interfaces between APL and real-mode routines called through the ⎕*NA* facility are TSR programs. Generally, these are small. Real-mode routines also require free DOS space when they are called. (See the Using ⎕*NA* chapter for details on real mode; see Note 9 for more information on free memory and real-mode routines.)

7. The session manager's communications buffer is used for buffered serial port I/O (COM1 or COM2) and terminal mode. The size can be set using the `C=` startup parameter. The range of sizes is 4 kilobytes to 63 kilobytes; the default is 4.

8. The Phar Lap Software 386 | DOS-Extender allows an 80386 system to run 32-bit protected mode programs under DOS. It runs from traditional DOS and supplies additional calls to 80386 protected mode. Many of the calls are emulated exactly as in DOS; some have been extended; many new calls have been added. You can control how much of the DOS-Extender is loaded into conventional memory by setting DOS-Extender switches (described in the following section).

   The DOS-Extender includes buffers for file I/O. You can control the amount of buffer space allocated using the `cfig386` utility. Using smaller buffers increases the frequency with which the DOS-Extender switches into real mode to call the DOS file services.

9. The free DOS memory region contains any memory that has not been consumed by other software. This space is used by *⎕CMD* and *)CMD*, and when a real-mode non-APL routine is called from APL using the *⎕NA* facility (see the Using *⎕NA* chapter).

   You can estimate the amount of free DOS memory by running *)CMD CHKDSK* and noting the report of available memory.

10. Expanded memory (EMS), as opposed to extended memory, keeps extra memory outside the real-mode address space. The only way you can use EMS is if a Virtual Control Program Interface (VCPI) provides the memory. The VCPI host automatically allocates all of the EMS that APL asks for as extended memory. Quarterdeck's QEMM and Qualitas' 386Max both conform to VCPI.

11. The APL workspace is allocated out of extended memory with a DOS-Extender allocation call, after allocating space for session manager screen memory (the default is `s=80`) and any graphics memory requested by `graphmem=`. Changing the amount of screen memory affects the maximum available workspace size. You can specify the workspace size with the `wssize=` startup parameter and vary it during the APL session using *)CLEAR*. If no

explicit size is specified, APL takes most or all of this region for workspace.

12. If space remains beyond the end of the workspace, it can be used by a protected-mode non-APL routine through the DOS-Extender call that allocates memory.

DOS-Extender Switches

The DOS-Extender bound into APL★PLUS II/386 has optional switches that you set with the `cfig386.exe` utility. Since `cfig386.exe` patches new settings into the APL interpreter, the switches are permanent — you only set them once. Be sure to set the switches in the working copy of your system.

The `cfig386` utility displays and allows you to set these switches:

- **`-CLEAR`**
  Clears all current switch settings and restores default DOS-Extender behavior. If you use this option, you must reset all options to the original settings on the APL★PLUS II disks.

- **`-EXTHIGH`** ***`hexaddr`***
  `-EXTLOW` ***`hexaddr`***
  Specifies the range of extended memory that APL can use to hold the interpreter and workspace. For example, `-extlow 200000h -exthigh 3FFFFFh` confines APL's use to the region between two megabytes and four megabytes. Set these options if a program (such as a virtual disk) uses extended memory but does not follow either the IBM/AT VDISK or the VCPI memory allocation convention. The DOS-Extender automatically avoids conflicting memory allocation for programs that follow either convention. If you know the range of addresses a nonconforming program uses, you can use these options to avoid conflict between it and APL. You can set these options to reserve memory that APL should not use; for example, to use DOS-Extender memory allocation calls within a non-APL subroutine. You should only use

these switches if you not running memory management software such as 386Max or QEMM.

- **-ISTKSIZE** *n*
  Controls stack sizes used by the DOS-Extender; default is 1. You may need to increase this option to 4 or more if VDI graphics calls are used to draw a large number of points per call. The maximum is 8.

- **-MAXIBUF** *n*
  Specifies maximum number of one-kilobyte blocks to allocate for buffers. Lesser amounts are allocated if the space is not available.

- **-MINIBUF** *n*
  Specifies minimum number of one-kilobyte blocks to allocate for data buffers for DOS system calls. Manipulating this parameter will not reduce space used.

- **-MAXREAL** *n*
  -MINREAL *n*
  Defines how much conventional DOS memory is left free by APL★PLUS II/386. Free conventional memory is needed during the APL session for ⎕*CMD* and)*CMD*, and for real-mode non-APL routines called with ⎕*NA*. Use one of these switches or the other; do not use both. -MINREAL specifies the minimum amount of conventional memory to leave free while -MAXREAL specifies the upper limit of the available memory. -MAXREAL FFFFh gives the most conventional DOS memory; -MINREAL 1000h gives the largest workspace. If you need a value in between, you should experiment with the switches to find the right value.

- **-NOPAG**
  Causes the APL★PLUS II/386 interpreter to run with 80386 demand paging turned off. This option circumvents a bug in early 80386 chips that cause a machine hangup when used with an 80387 in protected mode. Using this option may reduce the amount of workspace memory, since APL workspace sizes are always a multiple of 512 kilobytes. You cannot use the Phar Lap Virtual Memory Manager with APL or run APL under a virtual control program like 386Max or QEMM if you set this option.

- **`-VDISK`**
  The IBM/AT VDISK convention for allocating extended memory uses two signature blocks, one in conventional memory and one in extended memory. This option causes the DOS-Extender to pay attention only to the signature in extended memory. By default, the DOS-Extender consults both signatures and reports INSUFFICIENT MEMORY if either indicates that there is not enough room to run APL. Some DOS programs (such as the DOS version 3.3 PRINT command) corrupt the conventional-memory copy of the VDISK signature, which can prevent APL from running even when extended memory is available.

- **`-WIN30GROW`**
  When used with the `-WIN30LIMIT` switch, `-WIN30GROW` determines if APL★PLUS II/386 asks Windows 3.0 for memory beyond the initial block allocated by `-WIN30LIMIT`. If you do not use this switch and the initial block does not satisfy a request for memory, APL★PLUS II/386 fails and displays an error message.

- **`-WIN30LIMIT`** ***`nbytes`***
  `-WIN30LIMIT` ***`+nbytes`***
  This switch sets the initial amount of memory that APL★PLUS II/386 requests from Windows 3.0 before processing command line parameters and configuration files. This block must be large enough to hold the APL★PLUS II/386 interpreter. The *nbytes* parameter specifies the maximum memory size in bytes that APL★PLUS II/386 can reserve. If you specify *nbytes* with a plus sign (+), the total initial memory is *nbytes* plus the size of the APL★PLUS II/386 .EXE file. The default value is +2097152; that is, two megabytes plus the size of the .EXE file.

Access to the Operating System

The system function ⎕*CMD* and the system command)*CMD* allow you to execute DOS commands from within APL. For example, you need not end APL and return to DOS simply to list the files in a directory. When you use ⎕*CMD*, the amount of DOS memory available to run a program is substantially less than that available when APL is not running.

The)*CMD* command allows you to access DOS from an APL session. If you use the command without an argument, you enter DOS temporarily from APL. You can then enter any number of DOS commands. Type `exit` to return to APL.

If you specify a DOS command as the right argument, you still exit APL, but return as soon as the DOS command finishes executing.

The ⎕*CMD* function also exits APL temporarily to execute a DOS command specified in the right argument. If you use an empty vector as the right argument, the DOS command interpreter is loaded into available free DOS memory. You can then execute any number of DOS commands. Type `exit` to return to APL.

You can also press Ctrl-Shift-D to execute individual DOS commands instead of using)*CMD*.

Warnings: Never rename a DOS file that is tied by the suspended APL session. Files on your disk will be damaged if APL writes to a file that was tied by APL before being renamed within DOS. **Never** switch disks in a floppy disk drive containing tied files before)*CMD* or ⎕*CMD*. **Never** run any DOS program that fails to release all allocated memory upon exiting; for example, TSRs such as RESTIME.COM, PRINT.COM, MODE.COM, ASSIGN.COM, and so on. **Never** alter any keystroke file.

When ⎕*CMD* or)*CMD* returns to APL, tied files are checked to validate their existence. If a file is not found, the message *SOME FILE TIES LOST* is displayed.

While the DOS command interpreter is active, the keyboard reverts to the standard DOS keyboard. If an error occurs while a DOS command is being executed, DOS reports the error and control returns to APL.

In addition to ⎕*CMD* and)*CMD*, the system function ⎕*INT* enables you to access DOS and extended DOS interrupts from APL. It invokes an interrupt routine after optionally setting the CPU general registers. Its syntax is

```
        result ← registers ⎕INT interrupt
```

The left argument is optional. It sets the values of the CPU registers to be used upon entry into the interrupt routine specified in the right argument. Up to seven integers can be given in order for the EAX, EBX, ECX, EDX, EBP, ESI, and EDI registers.

The right argument is the **decimal** value of the interrupt to be invoked. See your operating system technical reference manual for the available interrupts and how to use them. Generally speaking, low-numbered interrupts up to 33 (21h) should carry over from DOS to the DOS Extender intact; for example, DOS Int 21h (⎕*INT* 33) will still invoke standard DOS function calls.

The result is a vector of eight elements, consisting of the seven integer values of the corresponding registers used as the left argument, followed by the Eflags register. (Refer to your 80386 manual for the interpretation of the Eflags register.)

Access to Non-APL Routines

APL★PLUS II/386 allows you to access non-APL routines in two ways: through the ⎕*NA* facility and through the system function ⎕*CALL*. ⎕*NA* is the more versatile method and supports several different compilers for both real mode and protected mode. See the Using ⎕*NA* chapter for a complete discussion of using ⎕*NA*. This section describes ⎕*CALL*.

The system function ⎕*CALL* allows you to execute assembly language programs from APL. To use this function, you must be familiar with 80386 protected mode assembly language and you must have a protected mode assembler and linker to assemble and link your code.

STSC recommends and supports the Phar Lap Software 386|ASM and 386|LINK packages for preparing code to use with ⎕*CALL*.

You should use ⎕*CALL* only if you are an experienced assembly language application developer. Programming in 80386 protected mode has several significant differences from assembly language programming in APL★PLUS PC.

You must understand the ramifications of multiple pointers to the same object, interactions between DOS facilities and the extended DOS facilities (protected mode), and the system services of APL★PLUS II/386 as described in the Interpreter Support Services appendix in the *Reference Manual* and in the VARINFO.TXT file.

A number of facilities within the APL interpreter are available through Int C8h; for example, the ability to create an APL variable or to determine a file handle.

Several assembly language source programs, such as SSTOMAT.ASM, are supplied on your distribution disks to help you understand this level of programming.

For details on ⎕*CALL* and its associated system function ⎕*STPTR*, refer to the System Functions, Variables, and Constants chapter in the *Reference Manual*. For details on the supplied interpreter support services, refer to the Interpreter Support Services appendix in the *Reference Manual*.

Customizing Your System

This section explains some of the ways you can customize the APL★PLUS System to suit your needs:

- you can redefine the keyboard
- you can customize for a non-IBM environment
- you can change default settings with *⎕POKE*
- you can build a custom Help file.

If you have a color monitor, you can also set certain colors in your system. See the "Using Color" section in the Managing Screen and Keyboard Data chapter for information on using color attributes with APL★PLUS II/386.

Redefining the Keyboard

One feature of APL★PLUS II/386 is the ability to redefine the keyboard. You can customize the default keyboard to correspond to foreign language keyboards, emulate the keyboard for APL★PLUS PC, or set up different keyboard layouts.

You can redefine the keyboard in two ways:

- Use the `key=` parameter in the configuration file to set up new keyboard definitions.

- Use the functions in the supplied *KEYBOARD* workspace to redefine the keyboard from APL.

How APL★PLUS II/386 Interprets Keystrokes

To redefine the keyboard correctly, you must understand how APL★PLUS II/386 interacts with the keyboard processor.

Note: The following discussion assumes that the default shift key definitions are in use, in which case the physical keyboard right and left Shift keys, Alt keys, and Ctrl keys produce shift actions. Which keys produce shift actions is controlled by the shift key table.

Each key on your keyboard has a **keybutton number**. (See the keyboard diagram in Appendix A of the *Reference Manual*.) When a key is pressed or released, a **scan code** is transmitted to the system. This code is hardware dependent. The system converts the hardware-dependent scan codes to keybutton numbers that are common to all machines.

First, the system finds out if a shift key has been pressed. To do this, it looks for the keybutton in the **shift key table**. This table includes a keybutton number, a raw shift state number, and the current state of the key (up or down). The **raw shift state number** specifies the type of shift key and is a power of two, as shown in Table 11-2.

The Shift, Ctrl, and Alt keys can be those on the left or right sides of your keyboard. In addition, Ctrl and Alt can be the plus (+) and minus (-) keys on the cursor pad when NumLock is off.

If the keybutton number is in the table of shift keys, the system remembers whether this key is now up or down.

Table 11-2. Raw Shift State Numbers

Raw Shift State	Default Shift Key
0	No shift
1	Shift
2	Ctrl
4	Alt

If a nonshift key is pressed, the raw shift state numbers for all shift keys that are currently pressed are combined in an "or" fashion. For example, if you press both the Shift keys and the Ctrl key, their raw shift state numbers are combined to result in 3.

The system now uses this number as an index into the effective shift state table, shown in Table 11-3. The value at the given index is the effective shift state.

Table 11-3. Effective Shift States

Raw Shift State	Effective Shift State	
	APL Keyboard	Text Keyboard
0 (no shift)	0	1
1 (Shift)	2	3
2 (Ctrl)	4	4
3 (Ctrl-Shift)	9	9
4 (Alt)	5	6
5 (Alt-Shift)	7	8
6 (Alt-Ctrl)	10	10
7 (Alt-Ctrl-Shift)	4	4

The keybutton number is combined with the effective shift state to form an effective keystroke. The effective keystroke is put into the type-ahead buffer.

Note: The shift-states returned by port 19 of ⎕*ARBIN* are the states in the left column in the preceding table (APL keyboard effective shift state).

When APL needs input from the keyboard, it requests an event. An event is a number in the range 0 to 32767 that represents a character or a keyboard action (such as a cursor movement). The routine that services this request first looks in the event buffer; that is, the buffer used by ⎕*INBUF*, which contains events. The type-ahead buffer contains effective keystrokes. If the event buffer is not empty, the system uses the next event from this buffer.

If the event buffer is empty, the routine picks the next effective keystroke from the type-ahead buffer, uses the effective shift state to choose which of several event tables to use, and looks up the keybutton number in the specified event table. The value at this location in the event table is the event returned to APL. If there are no events or keystrokes available, the system checks for mouse movements or button presses according to the setting of ⎕*MOUSE*.

Some events are performed immediately when the keys are pressed. These events are called immediate events, and include actions such as Break, Pause, and Reboot. Before an effective keystroke is put into the type-ahead buffer, the event table is checked to see if the event is a immediate event. If it is, the immediate event is performed and not put in the type-ahead buffer.

Defining Non-English Keyboards

The capability of redefining the keyboard makes it possible to set up keyboards that support non-English diacritical marks and accents.

To support foreign accent prefix keys, several accent prefix events have been defined. The accent prefix table contains one character for each of these accent prefix events. Because these differ with each language, you must define your own prefixes. For example, you may want Ctrl-' to produce a French accent grave when used with another letter. The table reserves event numbers 533 through 540 for your foreign accent definitions. Use the *ACCENTPREFIX* function in the *KEYBOARD* workspace to define your accent prefixes.

When APL receives one of these events in response to a request for input, it requests another character. It then takes this character and the character from the accent prefix table and attempts to form an overstrike from the two characters. If the two characters do not form a legal APL overstrike, APL beeps and cancels the accent prefix event; otherwise, the overstrike is used as the input character.

Overstrikes are defined in the **overstrike table**. Each entry consists of a triple: the character pair that can be overstruck and the character formed by the overstrike.

The foreign capslock table contains pairs of characters that are treated as uppercase/lowercase pairs when CapsLock is applied. (Note that CapsLock actually trades upper- and lowercase, rather than converting everything to uppercase.)

DOS Keyboard Handling

The normal APL★PLUS System keyboard routines "borrow" the keyboard interrupt vector from BIOS and drive the keyboard directly, bypassing BIOS and DOS. However, you may want to run the system using some or all of the normal DOS keyboard handling. The APL★PLUS System provides two modes that use the DOS function call interrupt to get a character:

- the fixed DOS keyboard
- the switchable DOS keyboard.

In the fixed DOS keyboard, the keyboard interrupt vector is not changed at all. The character returned by the DOS keyboard input function call is used as an index into the DOS keyboard conversion table. The value at this location is used as the combined raw shift state-keybutton number. The remainder of the processing is the same as the APL★PLUS System keyboard processing (starting with the raw-to-effective shift state conversion). To use this keyboard driver, use the session parameter env=2 in your configuration file. You cannot switch out of this mode during your session.

The switchable DOS keyboard is similar to the fixed DOS keyboard, except that you can toggle between this mode and the APL★PLUS II/386 keyboard by pressing Alt-F5. To use this mode, you cannot start APL★PLUS II/386 with the e=2 startup parameter. The keyboard interrupt vector is borrowed by APL★PLUS II/386. While the DOS keyboard is on, APL★PLUS II/386 passes incoming keystrokes to DOS, then uses the DOS keyboard input function call to get the keystrokes again after DOS has handled them. This behavior allows the APL★PLUS System to recognize the keyboard toggle keystroke even when the DOS keyboard is in effect.

Because the DOS keyboard ignores many keys when Ctrl, Alt, Ctrl-Shift or Alt-Shift are held down, certain shift prefix events have been defined. A key defined as a shift prefix key, when pressed, acts as if the corresponding shift-type key is held down for the next keystroke. For example, if the Tab key is defined as the Ctrl prefix, then pressing "Tab" and then pressing the letter "I" is treated as if the Ctrl-I was pressed.

Certain special keys on 101- and 102-key keyboards, like those on the dedicated cursor pad, transmit more than one scan code to the keyboard interrupt routine (which interprets which key is being pressed). These keys transmit a prefix hex E0 before the normal scan code. The 101-key keyboard table accommodates this prefix. The scan code following the E0 prefix is used as an index into the 101-key table, and the value at this location is used as the scan code.

Warning: Since this table deals with idiosyncracies of the 101- and 102-key keyboards and uses scan codes and not keybutton numbers, **do not** modify this table unless you understand how these keyboards work at the scan code level. Modifying this table can also invalidate the keybutton-to-scan code translation done by the distributed functions. For example

```
0 EVENT 76 463 ⋄ 1 EVENT 76 463
```

normally makes the Del key on the dedicated cursor pad delete the current line. It will not have this effect if location 83 of the 101-key keyboard table has been modified.

Defining Keys in the Configuration File

If you want to redefine the keyboard every time you use APL, you should use the key startup parameter in your APL configuration file. You can include as many new definitions as you want. The parameter takes the form:

```
key (1000⊥ess,kkk)=eventnumber
```

The *ess, kkk* parameter defines the effective shift-state and the three-digit keybutton number; the *eventnumber* parameter defines the event. For example, to use the keystroke combination Alt-I to toggle insert mode in the APL keyboard, you use:

```
key 5024=324
```

Shift-state 5 indicates Alt on the APL keyboard, 24 is the keybutton number for I, and 324 is the event number corresponding to toggle insert mode.

If you normally use the Text keyboard, you use:

```
key 6024=324
```

Shift-state 6 indicates Alt on the Text keyboard, 24 is the keybutton number for I, and 324 is the event number corresponding to toggle insert mode.

Defining Keys with the *KEYBOARD* Workspace

You can also redefine the keyboard during your APL session by using the functions in the supplied *KEYBOARD* workspace. These functions make it easy to both see and redefine the tables for foreign accent prefixes, effective shift states, events, overstrikes, and the DOS and 101-key keyboards. For details on using the functions in this workspace, see "Conversion and Customization Utilities" in the *Utilities Manual*.

Using APL in Nondefault Environments

APL★PLUS II/386 uses custom programming to perform input/output tasks faster or in a more versatile way than DOS. This custom programming cannot be used in a computer that does not match the IBM architecture on which the custom routine relies. The `Env=` startup parameter allows you to select or suppress the custom input/output programming to be used and rely on standard DOS calls instead. This section explains the available values for the parameter. The default is `0`, the IBM environment.

- **Env=2, Fixed DOS Keyboard**
  Using the DOS or BIOS routines for keyboard input allows many computers that are not IBM-compatible to work with APL★PLUS PC. By default, the e=2 startup parameter specifies the DOS keyboard calls. You can use the BIOS keyboard calls by setting:

  ```
        1 ⎕POKE 906
  ```

  Certain limitations apply when you use `e=2`. The BIOS keyboard calls are less restrictive than the DOS keyboard calls, and are the least restrictive if you use a computer with enhanced BIOS. This section lists all of the possible limitations. You must experiment with your computer to determine which ones affect you.

- ❑ You cannot enter composite APL characters with the Alt key in the APL keyboard. In the unified keyboard, any key accessed with the Alt-Shift combination is unavailable. Use Alt-Ins to create the following characters: ⍙ ⍝ ⍀ ⌿ ⍎ ⍫ ⍨ ⌹ ⍟ ≡ ⍕ ⍒ ⍋ ⌽ ⍉ ⊖ ● ⍱ ⍲ ⍞.
- ❑ You cannot use some of the control keys in the full-screen editor, the session manager, or terminal mode. Use the functions in the *KEYBOARD* workspace to specify other keys to perform these functions.
- ❑ The Esc key activates the Exit function, normally provided by Ctrl-Esc.
- ❑ In the DOS keyboard, Ctrl-C is a synonym for Ctrl-Break. In the BIOS keyboard, Ctrl-C still acts as the Copy function. When you press Ctrl-Break, the characters ^*C* and a newline character are sent to the screen.
- ❑ The operating system intercepts Ctrl-PrtSc and Shift-PrtSc, which may make them inoperable, depending on DOS. You can turn printing on (Ctrl-PrtSc) by storing the value 516 on a function key. You can print the screen (Shift-PrtSc) by storing the value 321 on a function key.
- ❑ Ctrl-↑, Ctrl-↓, and other Ctrl keystroke combinations are not recognizable DOS keystrokes and may not work.

- **Env=8, Beep**
  Some computers do not have sound-generating hardware that is completely compatible with the IBM PC family. These computers cannot use ⎕*SOUND* and the optional click for each keystroke. Selecting env=8 causes APL to use the BIOS call for its audible beep rather than accessing the audio hardware directly.

- **Env=16, COM1 RS-232 Serial Port**
  This parameter allows you to use the built-in DOS driver for the RS-232 port with the following behavior changes:

  - Data reception is not buffered, so your computer can only receive data when it is waiting. For example, data are lost if you switch out of terminal mode while data is being sent to your computer.

  - Sending Ctrl-S to a remote computer that recognizes this signal often results in data being lost between the time the Ctrl-S is sent and the time the remote computer stops sending data.

  - Data from a remote computer may be lost during communication faster than 300 baud unless the remote computer sends idles after each line and send no lines longer than 80 characters.

  Note: If you use a mouse attached to COM1, you can use this parameter to prevent APL from trying to initialize COM1. If you cannot use `e=256` when using a mouse, use `e=16`.

- **Env=32, LPT1 Printer Interface**
  This parameter allows you to use DOS printer interface routines. When you do, the APL★PLUS System cannot recover from attempting to print data when the printer is not ready. You must realize that the printer is not ready and make it ready, or press Break to regain control of the system.

- **Env=256, Switch Communications Ports**
  This parameter buffers COM2 instead of COM1 and makes COM2 the port for terminal mode. Only one RS-232 port can be buffered in one session; however, both ports will use XON/XOFF protocol.

  Note: If you use a mouse attached to COM1, you can use this parameter to prevent APL from trying initialize COM1. If you cannot use `e=256` when using a mouse, use `e=16`.

- **Env-1024, Soft Character Set Support**
  This parameter allows you to use one of the soft character sets supplied with the system with the following limitations.

  - ❑ You must have an IBM color graphics adapter or emulator, or a Hercules monochrome graphics adapter or emulator.
  - ❑ For Hercules graphics, you must use the `hercsoftchr=1` startup parameter when you start APL.
  - ❑ The only available non-default attributes are reverse video and underlining.
  - ❑ Screen output and updates are slower.

  When you use this parameter, you can press Alt-F7 to toggle between the soft APL character set and the hardware character set.

Using Peeks and Pokes

The system functions ⎕*PEEK* and ⎕*POKE* allow you to examine and change the settings of various options in APL★PLUS PC and APL★PLUS II/386. APL★PLUS II/386 has several ways to examine and change system options:

- You can customize system options in your APL configuration file. See the Getting Started chapter in this manual.
- You can change system options using the session manager menus during your session. See the Editing with the Session Manager chapter in this manual.
- You can set colors for certain aspects of the system. See the Managing Screen and Keyboard Data chapter in this manual.

- You can redefine the keyboard and set up programmed function keys using the *KEYBOARD* workspace and ⎕*PFKEY*. See the Managing Screen and Keyboard Data chapter in this manual and the "Redefining the Keyboard" section in this chapter.

By using these alternatives, you can avoid using ⎕*PEEK* and ⎕*POKE* in your applications. However, to maintain compatibility with APL★PLUS PC, APL★PLUS II/386 emulates most of its values for ⎕*PEEK* and ⎕*POKE*. The ⎕*PEEK* and ⎕*POKE* Values appendix in the *Reference Manual* is a list of these values.

Because of the 80386 hardware, however, you must restrict ⎕*SEG* only to an empty vector to use ⎕*PEEK* and ⎕*POKE* in the same way as in APL★PLUS PC. When you do, the right argument contains integers that correspond to certain ⎕*PEEK* and ⎕*POKE* locations in APL★PLUS PC. When you use ⎕*POKE* in this way, it chooses the system options that emulate APL★PLUS PC, but it does **not** actually modify the memory addresses.

When ⎕*SEG*←0, the right arguments to ⎕*PEEK* and ⎕*POKE* are 32-bit absolute machine addresses of the 80386 CPU. Only addresses in the first megabyte can be accessed in this way.

Building a Custom Help Facility

The APL★PLUS II/386 Help facility comes with one default Help file, SYSHELP.SF. The system accesses and displays the contents of this file whenever you start the Help facility. However, you can create your own help file and use it in place of the default file. This section explains how.

A Help file is an APL component file whose components contain delimited character vectors. The Help facility displays one help file component per screen. Each component has from one to six segments.

1. **Title**
   This is a character string to be centered at the top of the screen. A title segment with a single space displays as a blank line.

2. **Text**
   This is explanatory text you want to display. An empty segment uses no lines on the screen.

3. **Choice list**
   This is a list of choices the user can choose from. The Help facility displays the choices in columns in the remaining space before the footer line and highlights the first choice.

4. **Footer line**
   This is the text appearing at the bottom of the help screen. If this segment is empty, the system displays the default footer, which reads:

   "Esc = Exit   Alt-F10 = Back   Enter = Choice"

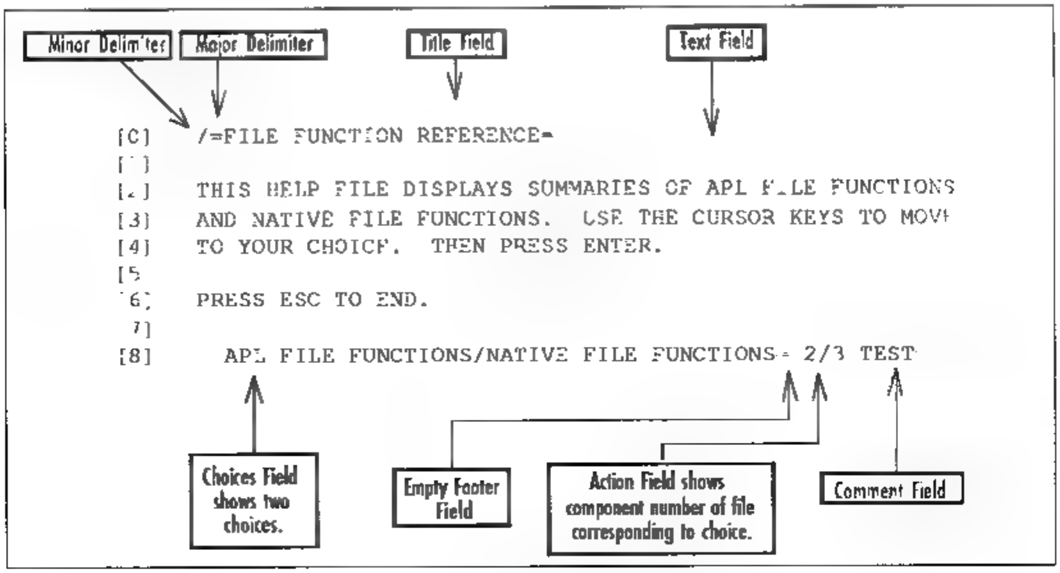
5. **Action list**
   This is a list of actions corresponding to the choices; another component number in the file to display when that action is selected. The action list is never displayed.

6. **Comment**
   This contains notes and other text for your own use. The comment is never displayed.

Any segment can be empty. To create an empty segment, place two major delimiters side by side with nothing in between. You can omit trailing empty segments; the system interprets missing segments as empty.

The first component in the file acts as a directory or index into the remaining components in the file. If the user specifies a topic when starting the Help facility or at anytime while they are using help, the system searches the first component, from the beginning, for a choice that matches the specified topic. The system then performs the corresponding action in the action list. See the "Designing the Directory" section in this chapter for information on designing and creating a directory.

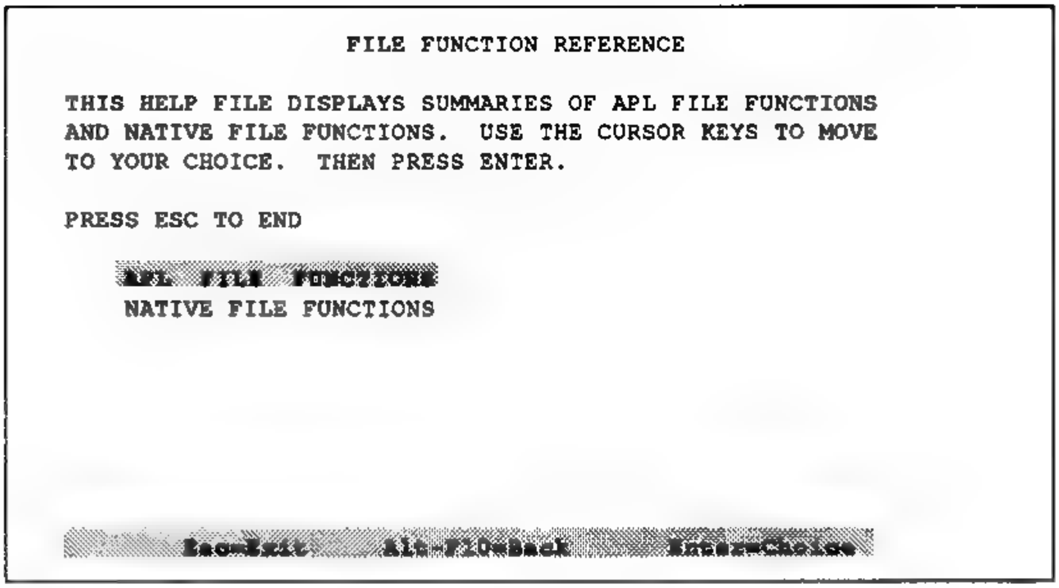
Figure 11-2 shows an example of a help file component and illustrates how you can use the full-screen editor to enter and edit the component segments. Figure 11-3 shows this same component as a help screen in the Help facility. For more examples of Help file components, examine the system Help file, SYSHELP.SF.



```
/=FILE FUNCTION REFERENCE=

THIS HELP FILE DISPLAYS SUMMARIES OF APL FILE FUNCTIONS
AND NATIVE FILE FUNCTIONS.  USE THE CURSOR KEYS TO MOVE
TO YOUR CHOICE.  THEN PRESS ENTER.

PRESS ESC TO END.

  APL FILE FUNCTIONS/NATIVE FILE FUNCTIONS= 2/3 TEST
```

Minor Delimiter
Major Delimiter
Title Field
Text Field
Choices Field shows two choices.
Empty Footer Field
Action Field shows component number of file corresponding to choice.
Comment Field

Figure 11-2. Help File Component



```
                    FILE FUNCTION REFERENCE

THIS HELP FILE DISPLAYS SUMMARIES OF APL FILE FUNCTIONS
AND NATIVE FILE FUNCTIONS.  USE THE CURSOR KEYS TO MOVE
TO YOUR CHOICE.  THEN PRESS ENTER.

PRESS ESC TO END

    APL FILE FUNCTIONS
    NATIVE FILE FUNCTIONS
```

Figure 11-3. Help Screen

A Help file delimited character vector begins with two distinct delimiters: the minor delimiter is first, followed by the major delimiter. The major delimiter marks the beginning of each segment. The minor delimiter forces multiple lines in segments 1 and 2 and separates the choices and actions in segments 3 and 5. You can use any two characters as the delimiters; however, these characters must be unique and cannot appear elsewhere in the character vector. For example, if a comma (,) appears in the text portion of the character vector, you cannot use it as a delimiter. The delimiters you choose can vary between components. You might use ⎕*TCNL* and the statement separator (⋄) as delimiters in one component and the delimiters backslash (\) and decode (⊥) in another.

Here are some examples of valid delimited vectors.

- This example shows a help file component with information only.

  ```
  '/=Title=This is text/and so is this.'
  ```

- This example shows a component with two choices.

  ```
  '/=Title=Choose one:=Commands/Functions==5/9
  ```

  The footer segment has been left blank. The actions for the choices are 5 and 9. When the user presses Enter, the contents of component 5 or component 9 will be displayed next.

- This example shows a component with a comment.

  ```
  '/=Title=Text====Need to add new features.'
  ```

Using Your Custom Help File

Once you complete your Help file, use the `help=` startup parameter in the configuration file to specify it to the system. If the Help file you specify is not in the current DOS directory, be sure to include the DOS pathname with the file name. The next time you start APL, this file is the one used by any call to the Help facility.

The following keys and mouse actions have predefined uses in the Help facility.

- Keys
  - **Esc**
    Exits the Help facility.
  - **Alt-F10**
    Backs up through the Help screens you viewed in reverse order.
  - **Enter**
    Performs the highlighted choice. If there is no choice on a screen, the system exits Help.

- **Mouse actions**
  - **Click left button on choice**
    Performs the highlighted choice.
  - **Click right button**
    Backs up through the Help screens you viewed in reverse order (same as Alt-F10).
  - **Click both buttons**
    Exits the Help facility. (On three button mice, click the two outside buttons.).

Designing the Components

This section contains hints on constructing your Help file and improving the way it works. To build your Help file, use the full-screen editor or the *HELPEDIT* function in the *SUBWNDWS* workspace.

Keep the components short. Short components have greater visual impact and are easier to read.

Be sure the component text fits on different types of monitors. If the text is too long to fit on one screen, use a single choice labeled "More" that leads to the next screen.

Similarly, if a choice list is too long to fit on one screen, break it into smaller lists or include a choice labeled "More Choices" that leads to the next screen.

You can also display lists as choices even if none of the choices lead further. A choice whose corresponding action is empty or all blank exits from the most local call to the Help facility.

Use the default footer where possible. It saves disk space and prevents your help file from becoming very large.

Always include a choice that returns the user to the "next highest" menu or screen. For example, on a submenu, include a choice that returns the user directly to the Main menu. Pressing

Alt-F10 backs the user up through the screens they viewed, in reverse order. Users may have to press Alt-F10 several times before returning to the submenu they want.

Line breaks in the full-screen editor represent newline characters in the component character vector. Like the minor delimiter, newline characters force the start of a new line in the text and title segments, but give undefined results in other segments. Use the newline character as the minor delimiter in large directories where the choice list or action list might cause a truncated line in the full-screen editor. You can also use the newline character in medium-sized components to number and align the choices and actions. To use the newline character as the minor delimiter in the full-screen editor, leave the top line in the editor blank and start the next line with the major delimiter. Use Ctrl-B to insert a newline character between two others, or use Alt-F3 or Enter.

Designing the Directory

The first component in the Help file acts as a directory to the remaining components in the file. By combining choices and actions in the first component, you can build many different paths through the Help file. Be sure to test your Help files thoroughly. Errors that occur during use of a Help facility are not handled by *⎕ELX* and *⎕DM*. Use the *HELPDIR* function in the *UTILITY* workspace to view the structure of your directory.

The first empty choice in the Choice segment stops the display of choices on the screen; however, the remaining choices are still available to the user. You can use these "hidden" choices to directly access topics in the Help file, bypassing any menus and submenus in the file. For example, to directly access information on *⎕FTIE*, include *⎕FTIE* in the hidden choices list and specify the appropriate component number in the action list.

You can also hide choices that are not for general use — such as extra components for application stewards. You can then tell those authorized the number or topic to enter to get to the hidden components.

If the user specifies a topic when starting Help or if they enter a topic while in Help, the system searches the directory, from the beginning, for the topic. Help selects the first complete or partial match it finds in the choice list and performs the corresponding action in the action list. If the system does not find a match, it displays an error message and returns to the calling environment. For example, suppose ⎕*CRL* and ⎕*CR* are two choices in the choice list and that ⎕*CRL* appears first in the list. If the user enters ⎕*CR*, the system will select ⎕*CRL* as the first match. In order for the system to select ⎕*CR*, it must appear first in the choice list.

You can include numeric choices in the directory, however, Help interprets numbers in one of two ways. If the user enters a number that is less than the number of choices in the directory, Help performs the action associated with that choice. That is, if there are 15 choices in the directory and the user enters the number 10, Help performs the action associated with the 10th choice in the directory. If the user enters a number that is greater than the number of choices in the directory, Help searches the directory for a number that matches the one entered. For example, if the 10th choice is 96 (for ⎕*PEEK* 96), entering 96 also performs the action associated with the 10th choice. In a large directory, you may not be able to use numbers by themselves in the choice list. You can either limit the size of your directory or include a non-numeric character with the choice — such as P96 instead of 96.

12 Using ⎕*NA*

The system function ⎕*NA* allows you to call external routines from within APL★PLUS II/386 as if they were APL user-defined functions.

This chapter explains how you use ⎕*NA* to call non-APL programs. It assumes that you already know how to write, compile, and link programs in other languages.

The chapter is organized as follows.

- "⎕*NA* Concepts and Use" explains how you use ⎕*MLOAD* to create a "module" and how you use ⎕*NA* to create an "associated function." It also explains how you use an associated function once you create it. You also learn more about the arguments and results of associated functions.
- "Real-Mode and Protected-Mode Routines" discusses the differences between real mode and protected mode, and describes the mechanisms the system uses to call routines in these modes.
- "Using Externally Resident Modules" describes the new facility for calling real-mode external routines.
- "Using the ⎕*NA* Tools" illustrates the use of the ⎕*NA* tools to analyze and construct APL values for external routines.
- "Tools for Building and Modifying External Routines" describes a suite of C routines that you can use to build and customize external routines that you can call from APL★PLUS II/386.

The important terms used in the chapter are:

- **External Routine**
  A program written in a language other than APL that has been compiled and linked.

- **Parameter**
  An argument to an external routine.

- **Associated Function**
  The function created by ⎕*NA* that calls your external routine.

- **Module**
  An APL variable created by ⎕*MLOAD* that contains the text and data of the executable non-APL program, along with other information necessary for calling and locating routines in the program. A module can comprise many routines, each of which has its own associated function.

- **Externally Resident Module,**
  An executable DOS-resident file that contains a suite of external routines. Unlike other modules, this type of module remains outside of the workspace. It does any necessary initialization and then calls APL★PLUS II/386.

- **Stub Module**
  A type of module that you use to call an externally resident module. Unlike other modules, this type of module does not contain executable code; rather, it contains descriptors that locate the entry points of the external routines in an externally resident module.

- **Language Interface Program**
  A terminate-and-stay-resident (TSR) program that provides the transfer mechanism between APL★PLUS II/386 and an external routine that runs in real-address mode.

- **Protected Mode**
  A mode of the 80386 processor that uses 32-bit addresses.

- **Real-address Mode**
  A mode of the 80386 processor that uses the 16-bit segmented architecture of the 8086. Throughout this chapter, **real mode** is used synonymously with real-address mode.

For related documentation on ⎕*NA* and ⎕*MLOAD*, see their descriptions in the System Functions, Variables, and Constants chapter of the *Reference Manual*.

⎕*NA* Concepts and Use

⎕*NA* defines the interface between APL★PLUS II/386 and a routine written in a compiled language by creating an associated function in the workspace. You can use ⎕*NA* to make utility routines available to APL applications. External routines may be useful runtime library routines or fast implementations of algorithms that are hard to express efficiently in APL.

The associated function behaves like a user-defined, locked monadic function in the workspace. You can save and load it with the workspace or copy it into another workspace.

When you use an associated function, APL★PLUS II/386 transfers control to the external routine and supplies parameters in a form that the external routine expects. The external routine then behaves as if it had been called from its native environment. When the external routine finishes execution, any results are then passed back to APL★PLUS II/386 through the associated function.

Using External Routines through APL★PLUS II/386

This overview outlines the general steps you take to use an external routine from APL★PLUS II/386. For description of how you use external routines that stay resident in DOS, see the "Using Externally Resident Modules" section later in this chapter.

1. Write, compile, and link your external routine. This chapter assumes that you know how to write programs in other programming languages and how to compile and link them to produce executable DOS files and load maps.

2. If you are using the original real-mode facility, load the appropriate language interface program into memory **before** you start APL★PLUS II/386 from DOS. You can load more than one interface if you are using external routines written in more than one language.

3. Start APL★PLUS II/386 and load the workspace you use to run your external routine.

4. Create a module with *⎕MLOAD*. The module is an APL variable that contains the executable file for your external routine.

5. Create an associated function with *⎕NA*. When you create the associated function, describe the external routine to APL, so that your associated function can pass the appropriate arguments in a form that the external routine expects.

6. Run the associated function. The associated function checks the module to be sure it contains the information necessary to run the external routine. The associated function converts and transfers the arguments to the external routine. The external routine runs as you called it from its normal environment. Any results from the external routine are passed back to the associated function and APL★PLUS II/386.

Creating Modules

Before you can use an associated function, you must create a module. A module is an APL integer variable that appears to be an ordinary integer vector in the APL workspace. You can save it in a workspace, copy it, or store it in a file. A module can have its own local data areas and global variables, if they are declared in the module's source language. These are preserved between calls to associated functions.

The monadic system function *⎕MLOAD* creates a module from the executable file and load map created when you compile and link your external routine. It reads a DOS executable file together with the map files created by a linking loader. It combines these files into a data structure consisting of a symbol table of entry points, and the executable code and data segments of the executable file.

A DOS executable file typically contains one entry point, where execution begins when the program is run in DOS. The file is formed by combining one or more object files with ordinary DOS-linking loaders. It can contain a number of individually named routines. When you load the file into APL with *⎕MLOAD*, the file becomes a library of individually callable routines. You can create a separate associated function for each routine in the executable file and call each as a distinct entry point.

When you create an associated function, the module does not need to be in the workspace. If you copy an associated function from a saved workspace, the module for that function is not copied automatically. You must copy the module explicitly.

If you want to associate an executable file with the workspace permanently, you can save the module in the workspace. To use the most recent version of an executable file, you can re-create the module by using *⎕MLOAD* again.

Because you can save modules in the workspace, along with other APL objects, modules can "move" from one address in memory to another. If your real-mode external routines depend on dixed memory locations, you should consider using externally resident modules. See the "Using Externally Resident Modules" section later in this chapter for detailed information on this technique.

Caution: The internal structure of a module is intentionally undocumented, since this structure is intended for system use only. Do not try to interpret or manipulate this structure; it may change in future releases.

Warning: Because APL recognizes a module as an ordinary integer variable, you can use APL primitives to manipulate it; for example, you can divide the module by 3. If you do manipulate the module, however, you destroy the code saved in it.

Syntax of ⎕*MLOAD*

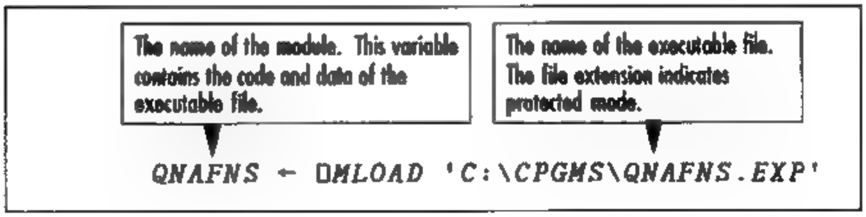
The syntax of ⎕*MLOAD* is

module ← ⎕*MLOAD* *'exefile,mapfile'*

exefile	The name of a linked, executable DOS file. You can omit the .EXE extension, but you must include the .EXP or .REX extension to use protected mode. If you do not specify a path, the current directory is assumed. (See "Real-Mode and Protected-Mode Routines' for information on protected mode and real mode.)
mapfile	(Optional) The name of the load map created when a .EXE file was linked. If you omit this argument, a file with the same name as *exefile* and the extension .MAP must exist in the same directory as *exefile*. The load map must exist for ⎕*MLOAD* to succeed with .EXE files, since they do not contain symbol tables that allow entry points within them to be found. .EXP files do not use load maps, but they **must** be linked with the `-symbols` option. .REX files also do not use the *mapfile*, since all of the necessary symbolic information is in the .REX file itself.
module	An integer vector that contains the executable program's code and data. The data within the module are not otherwise meaningful to APL.

Example of Using ⎕*MLOAD* to Create a Module

Figure 12-1 shows an example of using ⎕*MLOAD* to create a module.


The name of the module. This variable contains the code and data of the executable file.

The name of the executable file. The file extension indicates protected mode.

```
QNAFNS ← ⎕MLOAD 'C:\CPGMS\QNAFNS.EXP'
```

Figure 12-1. Creating a Module

In this example, the executable file is *QNAFNS.EXP*, which is in the directory *CPGMS* on disk drive *C*. The extension indicates that this file is a protected mode file. The module name is *QNAFNS*.

Creating and Using Associated Functions

To access the routines in a module, you must create an associated function for each routine in the module that you plan to call. The system function ⎕*NA* creates this function using a specific definition that you supply. ⎕*NA* creates a locked, monadic function that you can use as you would any other user-defined function.

The right argument to an associated function is a vector; typically, a nested array. Each item in the vector corresponds to one of the routine's parameters.

The explicit result is also a vector; typically, a nested array. The first item is the routine's explicit result (if defined), and the remaining elements are the output arrays, if any (arguments marked with "←" in the parameter list). These arguments may have been modified by the routine.

Syntax of ⎕*NA*

⎕*NA* has three possible syntax forms:

```
status ← 'defn' ⎕NA 'name'
status ← '' ⎕NA 'name'
defn ← ⎕NA 'name'
```

status	The explicit result. A 1 indicates that the function was created successfully; a 0 indicates the function was not created.
name	The name you want to use for the associated function.
defn	The definition of the external routine, including the language, type of result, name of the module, name of the routine, and the parameters expected by the routine. You can place ⎕*TCNL* characters between fields. You can include trailing comments in each line of the definition by preceding the comment with a ⍝ symbol. See the following section, "Constructing a Definition," for details.

Using ⎕*NA* dyadically with a left argument of *defn* creates an associated function. To find out if an associated function exists, you can use ⎕*NA* with an empty left argument:

```
status ← '' ⎕NA 'name'
```

The result is either a 1 or a 0. A 1 indicates that the right argument is the name of an associated function in the active workspace; a 0 indicates that it is not.

You can also use ⎕*NA* monadically to find out the definition used to create the associated function:

```
defn ← ⎕NA 'name'
```

Constructing a Definition

The definition of the external routine has the structure:

```
'@language ⊃result← module.routine (⊂parms)'
```

language The programming language used to write the external routine, or the externally resident module you used to start APL★PLUS II/386. If you specify a lanaguage, the routine must follow that language's subroutine linkage conventions.

This version of APL★PLUS II/386 supports the following language linkages:

- EXP
  32-bit protected-mode linkage following conventions of MetaWare High-C compiler and linked with the Phar Lap linker. This is the default output type from the Phar Lap linker.

- REX
  Protected-mode relocatable executable file with suffix .REX, as produced by the Phar Lap linker; for example, in conjunction with batch file F77TOREX.BAT provided with APL★PLUS II/386.

- MSC51
  Real-mode linkage following conventions of Microsoft C compiler versions 5.1 and version 6, and linked with a standard DOS linker.

- MSFOR50
  Real-mode linkage following conventions of Microsoft FORTRAN version 5.0 and linked with a standard DOS linker.

- TURBOC2
  Real-mode linkage following conventions of version 2 of the Borland Turbo C compiler; essentially the same as MSC, except for the way a function returns a floating-point result.

- *extmodule*
  The name of an externally resident module. See the "Using Externally Resident Modules" section later in this chapter for more information on this technique.

⊃result←	(Optional) The result (if any) returned by the external routine. The result uses the same datatype notation as *parm*. See the following section, "Datatypes Recognized by ⎕*NA*," for details. If the result has exactly one item, you can use a disclose (⊃) as a prefix to disclose the result.
module	The name of the module created by ⎕*MLOAD* that contains the routine or the stub module that points to the externally resident module.
routine	The name of the routine in the form in which the entry point would appear in the load map for the module. The distinction between upper- and lowercase is important. Some compilers insert a leading underscore (_) in the name of the routine. Check the load map (a file with suffix .MAP produced by the linker) to ensure that you are specifying the correct name.
parm	The form of arguments to be passed to the routine. Enclose the list of argument specifications in parentheses and separate them with commas. If the routine does not require parameters, you must still supply an empty list enclosed in parentheses. Each *parm* describes the datatype of each argument, how it

is passed, and whether the routine modifies it. If a *parm* is marked with a "←" as being "modifiable," it is an output array that is returned as part of the routine's explicit result, regardless of whether it actually was modified. If the argument consists of exactly one item, you can precede the specification with an enclose symbol (⊂) to enclose the item. See "Datatypes Recognized by ⎕*NA*" for details on datatypes; see "Passing Parameters in ⎕*NA*" for details on how parameters are passed.

Datatypes Recognized by ⎕*NA*

Datatypes are declared explicitly in compiled languages. Routines work correctly only if you pass parameters of the declared type. When you describe the result and parameters of the external routine in the syntax of ⎕*NA*, you must specify the correct datatypes and tell ⎕*NA* how to pass the arguments.

When you call the associated function, APL ensures that each of the arrays being passed has the appropriate datatype. If necessary, APL changes the datatype to make it conform to that expected. This process is called coercion. APL coerces datatypes by making a copy of the data and changing the internal representation in such a way that the value stays the same. It then passes the copy to the external routine. Since many real-mode routines require 16-bit integer data, APL generates temporary parameters to associated functions in this form — even though 16-bit integers are not used by the APL★PLUS II/386 interpreter.

For example, the variable *A* contains a Boolean array; that is, each number uses one bit per element in the computer.

```
      A
1 0 1 1
```

Suppose the external routine you want to use works with four-byte integer data; that is, each number uses four bytes per element in the computer. If this routine receives *A* in Boolean form as an argument, it fails because it expects each element of the array to be in 32-bit integer form.

When you call the associated function, APL★PLUS II/386 makes an integer copy of *A*, changes each one-bit element into a 32-bit 0 or 1, and passes the copy to the external routine.

A *DOMAIN ERROR* occurs if APL cannot coerce an array to the correct datatype without changing its values. Boolean and integer data can always be promoted to floating point, but floating-point values are converted to integer only if they are exact integers or within APL's "system fuzz" tolerance of whole integers. (See the APL Language Reference chapter in the *Reference Manual.*)

APL★PLUS II/386 does not use 2-byte integers in ordinary variables in the workspace. APL creates such values temporarily to provide them to the associated function, but any results returned are converted back to 4-byte integers.

Table 12-1 lists the available datatypes.

Table 12-1. Datatypes

Datatype	Description
*B*1	Boolean (1 bit per element in byte-wise order, most significant bit first)
*C*1	Character (1 byte per element)
*I*2	2-byte signed integer*
*U*2	2-byte unsigned integer
*I*4	4-byte signed integer
*F*8	8-byte double precision IEEE floating point
*G*0	General object; protected mode only. A variable in the internal form used by APL. Passed by reference. **Warning:** You must know the internal structures in the workspace, which are subject to change in future versions. For a description of the array structure, see the file VARINFO.TXT on the distribution disk.
*X*0	External representation. Argument is placed in dynamically allocated memory outside the workspace and passed by reference. Nested arrays are represented in linear contiguous form, with subarrays following higher-level arrays. Pointers to these arrays are represented as 4-byte offsets from the beginning of the array. *X*0 results can be of any type, shape, or depth. Used in protected-mode .REX modules or externally resident modules.

Specifying Parameters in ⎕*NA*

In addition to specifying the datatypes of the parameters, you must specify how to pass them to the external routine.

Compiled languages usually provide several distinct methods for passing parameters, such as by name, by value, and by reference. Pass-by-value and pass-by-reference are the most common. Pass-by-value is used to pass a single scalar element. Pass-by-reference is used to pass the memory address of the data rather than the actual data. This method is appropriate for passing arrays of arbitrary size, because only the starting address is passed. Pass-by-reference also allows a subroutine to modify the values stored at that address. ⎕*NA* allows you to pass scalars by value. You must pass arrays of more than one element by reference.

To indicate how you want the arguments to be passed, add a prefix to the datatype. Table 12-2 shows the possible prefixes.

When you call an external routine, APL★PLUS II/386 constructs an argument list from the right argument supplied to the external routine. The argument list consists of parameters that contain either the data value itself (pass-by-value) or an address of the data (pass-by-reference). The form in which arguments are passed varies among languages; APL★PLUS II/386 sets up the arguments in the form required by the language being used.

Table 12-2. Prefixes for Parameter Passing

Parameter	Description
⍑	Default. Passed by value. The actual value is passed to the external routine. Only one-element simple arrays can be passed by value.
*	Passed by reference. The machine address of the value is passed to the external routine.
∆	Reserved for future enhancement.
⍉	Passed by reference in column-major order as expected by FORTRAN; must be used instead of *. For arrays with ranks greater than 2, only the last two coordinates are transposed.
●	Forced coercion. Forces a modifiable copy of the argument item to be made. The modified argument is not returned in the result.

Arrays in APL workspaces are represented by data structures that contain a variety of descriptive and housekeeping data about the array, such as the shape and datatype, in addition to the actual data. Generally, only the data items themselves are meaningful to the external routine.

If the argument is passed by reference, APL passes the address of the first data value of the corresponding item of the argument. For example, if a routine has two arguments, both passed by reference (*I4, *C1), it requires a two-item nested array as its argument:

```
      ARG ← (99 100 101 102) ('ABCDEF')
```

The external routine is passed two arguments: the address in the workspace of the value 99 in 1⊃ARG, and the address of the 'A' in 2⊃ARG.

Figure 12-2 shows how APL passes an array by reference.

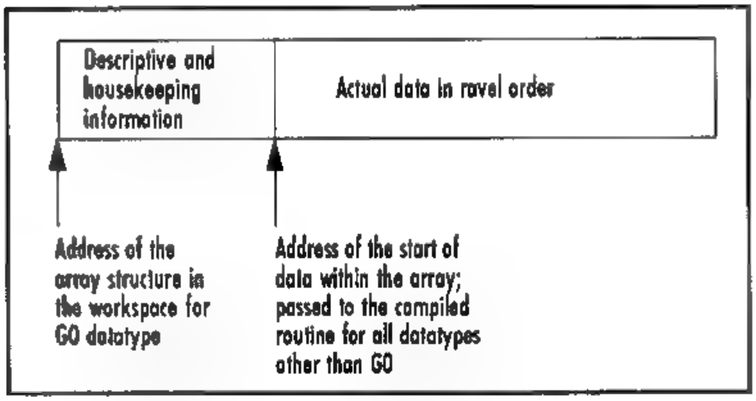

Descriptive and housekeeping information
Actual data in ravel order
Address of the array structure in the workspace for G0 datatype
Address of the start of data within the array; passed to the compiled routine for all datatypes other than G0

Figure 12-2. Passing an Array by Reference

Example Using ⎕*NA* to Create an Associated Function

This section describes how you use ⎕*NA* to create an associated function.

The sample external routine shown below is a Metaware High-C program. This routine, named `isum`, adds a vector of numbers and returns the sum.

```
int  far isum ( input, length )
int  far *input; /*address of first number in vector */
int length;        /*length of the vector */
{
int i;
int sum = 0;
        for   ( i=0;  i<length; i++ )   {
                     sum += input[i];
       }
        return(sum);
}
```

This routine requires two parameters:

- `input`
  The machine address of the first number in the vector, which is passed by reference.

- `length`
  The length of the vector, which is passed by value.

Before you can run `isum`, you must create the module from the compiled and linked C program:

```
ISUMOD ← ⎕MLOAD 'C:\CPGMS\ISUM.EXP'
```

Figure 12-3 shows how you use ⎕*NA* to create the associated function *SUMIT*

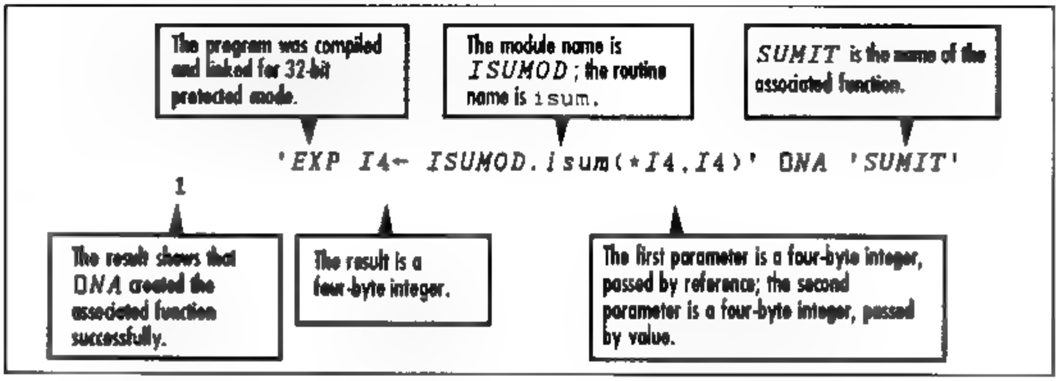

The program was compiled and linked for 32-bit protected mode.
The module name is ISUMOD; the routine name is isum.
SUMIT is the name of the associated function.
'EXP I4← ISUMOD.isum(*I4,I4)' ⎕NA 'SUMIT'
1
The result shows that ⎕NA created the associated function successfully.
The result is a four-byte integer.
The first parameter is a four-byte integer, passed by reference; the second parameter is a four-byte integer, passed by value.

Figure 12-3. Creating an Associated Function

Running Associated Functions

Once you create an associated function, you can run the non-APL function from APL★PLUS II/386; for example:

```
      X←⍳4
      SUMIT (X) (⍴X)
10
```

Although the module does not need to be in the workspace when you create the associated function, it **must** be in the workspace when you run the associated function. When you call an external routine, APL★PLUS II/386 follows these steps:

1. Checks the right argument to make sure that it is a scalar or vector whose length matches the number of parameters that you specified with ⎕*NA*.

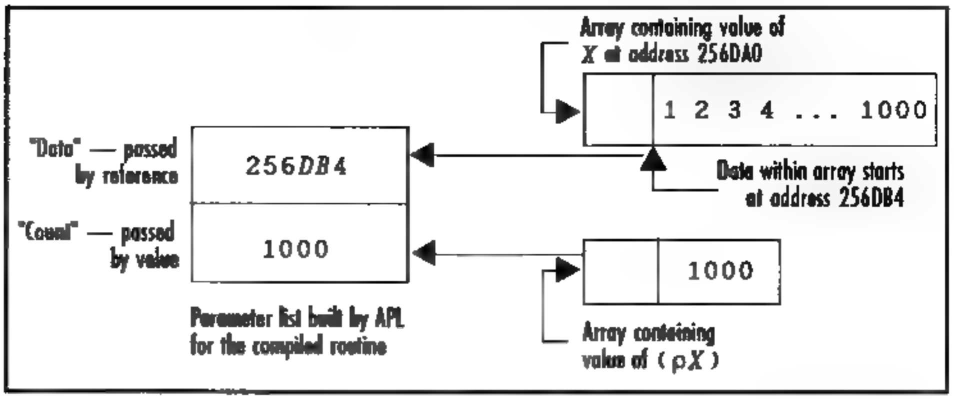
2. Locates the entry point in the module. If the module does not exist, APL produces a *MODULE NOT FOUND* error. If the internal structure of the module is not well-formed, APL produces an *INVALID MODULE STRUCTURE* error. If the module's symbol table does not contain the entry point, APL produces an *ENTRY POINT NOT FOUND* error.

3. Creates temporary copies of the parameters marked as "modifiable" and passes the parameter list. See Figure 12-4.

   Creates a temporary copy if the data in the APL array is not already of the type expected by the routine, but can be coerced to that type. Parameters are passed as specified in the associated function.

4. Calls the external routine.

5. When the external routine returns, builds an APL variable to contain the explicit result and returns it as the explicit result of the associated function.



Array containing value of X at address 256DA0
1 2 3 4 ... 1000
"Data" — passed by reference
256DB4
Data within array starts at address 256DB4
"Count" — passed by value
1000
1000
Parameter list built by APL for the compiled routine
Array containing value of (ρX)

Figure 12-4. Parameter List

Explicit Results and Output Arrays

The explicit result, if any, of an associated function is a vector, typically a nested array. The number of items depends on the description used with ⎕*NA* to define the associated function. If the routine returns a result, that result is the first item of the array. The remaining items of the explicit result consist of any of the arrays marked as modifiable output arrays with "←" in the ⎕*NA* left argument.

APL★PLUS II/386 makes temporary copies of all parameters marked as modifiable, and passes the copies to the routine. An external routine can safely modify the values in these copies without unexpected side effects.

When the external routine completes, APL★PLUS II/386 forms the associated function's result from the explicit result of the routine followed by the parameters marked as output arrays. If any output arrays are passed by reference, they are returned as enclosed items of the result.

If you do not specify a result or output arrays, the associated function returns an empty numeric matrix.

Output arrays allow an external routine to return one or more entire arrays to APL, rather than a single scalar. For example, the following external routine reverses the text in a character vector. It expects three parameters — the address of the input vector, the address of the output vector, and the length.

```
      'EXP ⊃TXTPGM.REVTXT(*C1,*C1←,I4)' ⎕NA 'REVERSE'
1
```

The second parameter is marked as modifiable. APL creates a temporary copy and passes the copy to the external routine. When APL★PLUS II/386 calls *REVERSE*, the second parameter must be the array that it will modify to contain the reversed character vector. The simplest way to call this routine is to pass the same array as both arguments:

```
      TXT ← 'THE INPUT UNMODIFIED'
      TXTR ← REVERSE (TXT)(TXT)(⍴TXT)
      TXTR
DEIFIDOMNU TUPNI EHT
```

```
      TXT
THE INPUT UNMODIFIED
```

Because the second argument to the routine is marked as an output array, a copy of *TXT* is created and passed. *REVERSE* then places the output in this copy. This copy is the same size as *TXT*, which is exactly what the routine requires.

Alternatively, you can create the output array explicitly:

```
      TXTR ← REVERSE (TXT) ((ρTXT)ρ' ') (ρTXT)
```

When the second parameter is **not** specified as modifiable, APL passes the address of the original array rather than making a copy.

```
     'EXP ⊃TXTPGM.TXTMOD(*C1,*C1,I4)' ⎕NA 'REVERSE'
1
      SPARETXT ← TXT ← 'THE INPUT UNMODIFIED'
      REVERSE (TXT)(TXT)(ρTXT)
      TXT
DEIFIDOMNU TUPNI EHT
      SPARETXT
DEIFIDOMNU TUPNI EHT
```

Since no copy was made of the input variable, *TXT* and *SPARETXT* refer to the same array in the workspace and the modifications done by *REVERSE* are to this shared array and possible other arrays in the workspace.

Caution: Be sure to indicate modifiable arrays when you define associated functions. APL★PLUS II/386 uses an array strategy based on reference counts, which means that a single array can have multiple references; for example:

```
      A←B←C←⍳1000
```

The array ⍳1000 is created only once. The variables *A*, *B*, and *C* point to that one array. If you pass *A* by reference and do not define it as an output variable, an external routine could modify the contents of *A*. However, *B* and *C* will also be modified.

As a rule, you should **never** modify the value of an array unless you mark it as modifiable or you are **certain** that it is referenced only as the variable you intend to modify. If you fail to follow this guideline, you can cause system crashes, data loss, or other unpredictable behavior.

Using Multiple Associated Functions

An associated function cannot create a new array in the APL workspace. You may need to create more than one associated function to handle such needs. This section presents an example of this technique.

The example is a utility for converting a delimited character vector into a matrix. This utility requires two distinct routines: one to determine the shape of the matrix that is produced, and one to transfer the delimited strings into the rows of the result matrix that is created with the required shape. The routines have the following C syntax.

```
far  vectomat1(cv,count,rows,cols)  /*  determine  shape  of  result
               matrix*/
char far *cv;         /* delimited character vector */
int count;            /* length of cv */
int far *rows;        /* how many rows needed (set by this routine) */
int far *cols;        /* how many columns needed (also set here) */
...
far  vectomat2(cv,count,rows,cols)  /*  move  strings  into  matrix */
char far *cv;         /* delimited character vector */
int count;            /* length of cv */
char far *result;     /* destination character matrix */
int rows;             /* shape ... */
int cols;             /*   of result */
...
```

A single module, *VECTOMAT*, contains both routines. ⎕*NA* creates the two associated functions:

```
'EXP VECTOMAT.vectomat1(*C1,I4,*I4←,*I4←)' ⎕NA 'VM1'
'EXP VECTOMAT.vectomat2(*C1,I4,*C1←,I4,I4)' ⎕NA 'VM2'
```

You then write the complete utility function:

```
    ∇MAT ← VECTOMAT VEC;ROWS;COLS
[1]  (ROWS COLS) ← VM1 (VEC(ρVEC) (0) (0)
[2]  MAT←⊃VM2 (VEC)(ρVEC)((ROWS,COLS)ρ' ')(ROWS)(COLS)
    ∇
```

Exiting the first routine to create an array in APL may seem inefficient, but the effect is actually minimal. The reshape primitive is very fast. Only reshape's initialization of the values in the result array is superfluous, but APL★PLUS II/386 performs this operation using a tight assembler routine. The overhead of forming the nested array arguments and calling the associated functions is very slight.

Real-Mode and Protected-Mode Routines

APL★PLUS II/386 can call routines that are written for two modes of the 80386 processor: real mode and protected mode. This section explains:

- real and protected modes
- advantages of protected mode over real mode
- how external routines are called in each mode.

The 80386 processor is capable of operating in several distinct modes. In "real-address mode," or real mode, the 80386 can execute the same object code as the 8088/8086 processor family. Programs written for real mode can run on any PC-compatible machine, whether the processor is an 8088, 8086, 80186, 80286, or 80386. Most software written for the PC, including APL★PLUS PC, operates in real mode.

In "32-bit protected mode," or protected mode, the 80386 operates as a 32-bit computer with a virtual (that is, memory-mapped) address space. The 80386 and its successors are capable of 32-bit mode; the 8088 and 80286 processors are not. APL★PLUS II/386 runs in protected mode.

The 80386 processor has additional modes, including 80286 protected mode and virtual-8086 mode, but they cannot be used from APL★PLUS II/386.

Associated functions allow APL★PLUS II/386 to call both real-mode and protected-mode routines. The choice of mode depends on the requirements for the external routine and the development tools available.

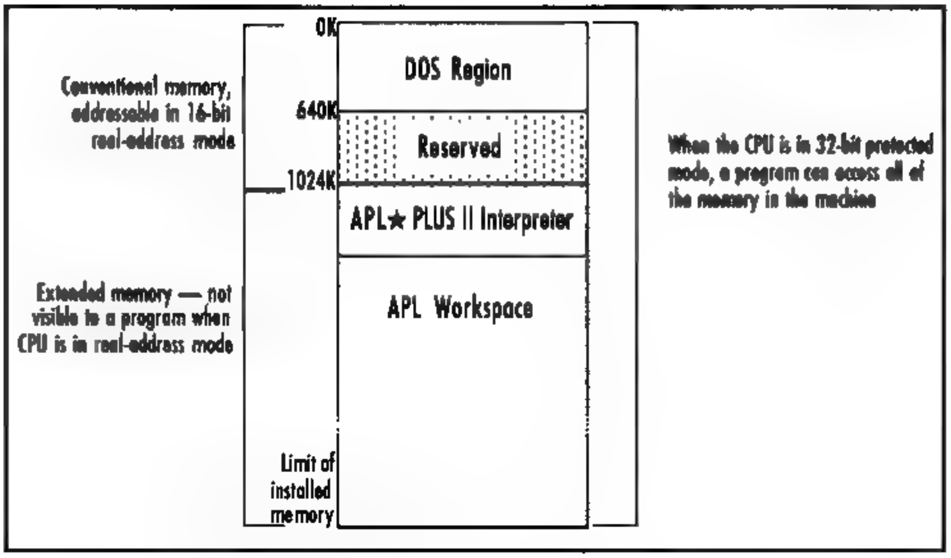
The difference between real mode and protected mode is the range of memory addresses available and the way they are addressed.

In real mode, memory is addressed using 16-bit segment and offset registers, and the total address space is 1024 kilobytes.

The normal MS DOS environment occupies this region, with addresses below 640 kilobytes available to DOS applications.

Protected mode can address any memory in the computer, including the extended memory with addresses above 1024 kilobytes. The APL★PLUS II/386 interpreter and workspace are located in extended memory.

Figure 12-5 shows a simplified memory map of an 80386 computer running APL★PLUS II/386. The figure shows the memory regions available to real-mode and protected-mode programs. (For more details on memory use, see Figure 11-1 in the Advanced Techniques chapter of this manual.)



Conventional memory, addressable in 16-bit real-address mode
Extended memory — not visible to a program when CPU is in real-address mode
0K
640K
1024K
Limit of installed memory
DOS Region
Reserved
APL★ PLUS II Interpreter
APL Workspace
When the CPU is in 32-bit protected mode, a program can access all of the memory in the machine

Figure 12-5. Memory Map of an 80386 Computer

Using Protected Mode

Protected-mode routines are those written, compiled, and linked in languages that take advantage of memory beyond one megabyte. Protected-mode routines have a number of advantages over real-mode routines:

- They can use the full 32-bit capacity and speed of the 80386 machine.
- They can "see" into the APL★PLUS II/386 workspace, because they are capable of addressing the memory where the workspace resides. APL, therefore, does not need to copy arguments and programs into the lower 640-kilobyte region as with real mode
- They can operate on arrays larger than 64 kilobytes.
- They do not have to compete for the limited space available in the 640-kilobyte region. (See Chapter 11 for a discussion on managing the memory in this region.)

Protected mode provides for virtual addressing. The processor maintains a table of defined memory segments. Each segment is mapped onto a region of real memory and denoted by a segment selector, which is an index into the segment table. This arrangement provides a 48-bit address space, consisting of a 16-bit segment selector and a 32-bit offset.

.EXP and .REX Files

The Phar Lap 386|LINK utility can produce output files in two formats: .EXP and .REX.

- **.EXP**
  Files with the suffix .EXP, the default for the Phar Lap linker, are similar to the .EXE files of real-mode DOS. They are designed to be loaded into memory and started with the 80386 code and data segments initialized appropriately. APL★PLUS II/386 can call routines in .EXP files by making

a far calls to those routines. (See "Far Calls and .EXP Modules," in this section.)

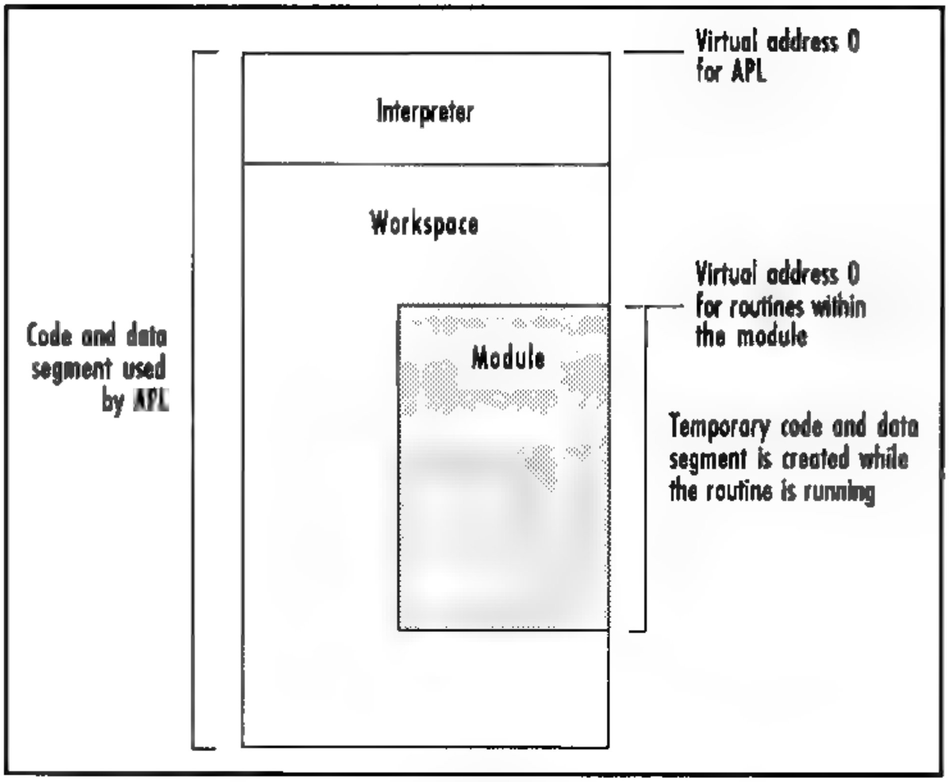
- **.REX**
  Files with the suffix .REX are a special form of executable file produced with a Phar Lap linker option. These files contain relocation tables that allow all of the memory references in them to be adjusted to reflect the address of the module in memory. APL★PLUS II/386 requires .REX file format for protected-mode FORTRAN routines produced by the NDP FORTRAN compiler. The batch file F77TOREX.BAT provided with APL★PLUS II/386 compiles a protected-mode FORTRAN program into a .REX file.

How .EXP Modules Are Called

Since the 32-bit offset is sufficient for the needs of current PC software, protected-mode compilers generally compile for this "small" memory model. The code ignores the segment register and uses the 32-bit offset registers for machine addresses. The assumption is that the 80386 data and code segment registers are set when the program is loaded into memory.

APL★PLUS II/386 uses this property to call .EXP protected-mode routines that have been loaded into modules. When a routine is called, APL creates a new memory segment in the CPU's segment tables, beginning at the origin of the 32-bit address space used by the compiler. The new segment overlaps the APL interpreter's address space and corresponds to the physical memory that contains the APL variable that holds the module. Routines within the module are then called with a far call, which sets the code and data segment registers to the new segment. The compiled routine runs in its own address space. Figure 12-6 shows this behavior.

This version of APL★PLUS II/386 supports .EXP files produced by the MetaWare High-C compiler for C language programs linked with the Phar Lap linker. Other compilers may be suitable if they use the same subroutine linkage as the MetaWare compiler.



Virtual address 0 for APL
Interpreter
Workspace
Virtual address 0 for routines within the module
Code and data segment used by APL
Module
Temporary code and data segment is created while the routine is running

Figure 12-6. Calling a Protected-Mode Routine from a .EXP File

How .REX Modules Are Called

A module loaded from a .REX file contains internal information that allows all of the machine addresses in the module to be found. When APL★PLUS II/386 first calls an entry point in a .REX module, all of the addresses in the module are adjusted to reflect the address in the workspace of the variable containing the module. If the variable subsequently moves to a different address because of a storage compaction or saved-workspace load, the addresses are automatically updated on the next call.

Because a .REX file runs from the same code and data segments as the APL interpreter itself, entry points in such a module are called with a near call and arrays are passed by reference with a near pointer. .REX programs also use near pointers for both internal C variables and arguments passed from the workspace. This makes it easier to integrate library routines into modules that require near pointers.

This use of near pointers is a disadvantage when debugging. Because the code and data segments are shared, errant pointers that were supposed to be pointing inside the .REX module can run off the end and destroy the workspace without causing a memory protection fault.

Because storage compactions ("garbage collects") are rare in typical applications of APL★PLUS II/386, the execution time needed to relocate a .REX module dynamically is generally not a cause for concern.

Sample Protected-Mode Routine

This section contains an example of how you write, compile, and run protected-mode routines. All of the example routines in this section are in files on the APL★PLUS II/386 distribution disks. To compile and link these protected-mode routines, you need the MetaWare High-C 386 compiler and the 386|LINK utility that is part of the Phar Lap 386|ASM/Link package.

The file `psum.c` contains two routines: `isum()` and `fsum()`. This file has no `main()` program, since the routine runs only from APL. The `isum()` routine is:

```
int _far isum(a, n)        /* sum an integer vector */
int _far *a;                /* address of the first item */
int n;                      /* number of elements to sum */
{
        int z = 0;
        while (--n >= 0) z += *a++;
        return(z);
}
```

The "_far" qualifier is an artifact of the strategy that APL uses to call .EXP modules; it is not used for .REX files. It indicates that the routine is called with a "far" call that sets both the code and data segment and instruction pointer registers.

The following command compiles the file `psum.c` for protected mode using the High-C compiler.

```
   c>hc386p psum -profile hc7.pro
```

(The file `hc7.pro` sets the option to turn the coprocessor on or off.)

The resulting file, `psum.obj`, can be linked using the Phar Lap linker:

```
   c>386link pstack psum -twocase -symbols
```

The `-twocase` option preserves the distinction between upper- and lowercase in the source file. The `-symbols` option includes the symbol table in the output file. The symbol table is required by APL to find the entry point within a module. (The inclusion of `pstack` in the load is explained in "Stack Segment" in the following section, "Technical Specifics for Protected-Mode Routines.")

The output from the linker is a protected-mode executable file named `psum.exp`. You can load this file into a module in the APL workspace with *⎕MLOAD*, and then define an associated function for `isum()` with *⎕NA*.

```
      PSUM ← ⎕MLOAD 'PSUM.EXP'
      'EXP I4←PSUM.isum(*I4,I4)' ⎕NA 'SUMIT'
1
      X←⍳1000 ⋄ RESULT←SUMIT (X)(⍴X) ⋄ RESULT
500500
```

Because the routine is compiled for a 32-bit computer, the size of an "int" is four bytes. In a 16-bit C compiler, you must declare the same variable as "long" to force the generation of code for four-byte integers. And, because the routine is compiled for protected mode, it works on arrays larger than 64 kilobytes.

The `isum()` routine illustrates the trade-offs of speed versus generality that are typical of external routines. The associated function *SUMIT* is faster than the APL operation of plus reduction (+/) in summing an integer vector. However, `isum()` lacks the logic that APL uses to detect when a calculation overflows the 32-bit integer; for example:

```
      +/2ρ1E9 ⋄ SUMIT (2ρ1E9) (2)
2000000000
2000000000
      +/3ρ1E9 ⋄ SUMIT (3ρ1E9) (3)
3000000000
¯1294967296
```

SUMIT returns a wrong answer because the computation overflows the 32-bit maximum for a positive integer. The +/ operation detects the overflow and automatically produces the correct answer in floating-point format.

Technical Specifics for Protected-Mode Routines

- **Far Calls and .EXP Modules**
  In the sample C program `psum.c`, the modifier `_far` is used for the entry point and pointer argument `a`, which is passed by reference. This modifier means that the routine expects to be called with a far call and that pointers are passed as far pointers.

  The far call is required in the example because of the way that the High-C compiler and Phar Lap linker are used to produce .EXP files. Most 32-bit 80386 programs, including APL★PLUS II/386, are compiled using the "small" memory model in which the code segment register CS is set once. Only the instruction pointer EIP varies during execution. Since this provides a 32-bit address space, the "small" model is large enough for any program that could be written for current microcomputer technology.

  The module loaded into the workspace with *⎕MLOAD* is also compiled for the small model, with all addresses that it uses based on the start of the module. Because modules are ordinary APL variables, they can move around in the APL workspace. When APL calls a routine in a protected-mode module, it creates a new logical segment in the 80386 segment descriptor tables based on the current address of the module. APL then makes a far call into the module with the CS register set to that segment selector and the EIP register set to the entry point.

The `_far` qualifier causes the High-C compiler to generate a far return, which restores CS to the code segment used by the APL interpreter. If CS is not restored, APL crashes as soon as the routine returns. (**Note:** Use of the `_far` qualifier may not be documented in the MetaWare High-C manuals; it is known to work in version 1.4 or later.)

For the same reason, the `_far` qualifier is required for all addresses that APL passes to an associated function. APL variables that are passed by reference lie outside the module's logical segment and cannot be accessed using the same segment register. Therefore, parameters passed from the workspace are declared far, meaning that the compiler generates code to set both segment and index registers when accessing them. Protected mode routines can thereby access any memory in the machine, notably arrays that have been passed by reference from elsewhere in the workspace.

- **Floating Point**
  The High-C compiler, like most PC compilers, provides several strategies for handling floating point. The default case generates code that works on machines with or without a math co-processor. Each floating-point operation generates a call to a subroutine that uses the co-processor if present and an emulation routine otherwise. The compiler can also generate inline 80387 instructions, which optimizes performance if a co-processor is present and fails completely otherwise. The inline code option is used to produce the APL387.EXE variant of the APL★PLUS II/386 interpreter.

- **Code Requiring Co-processor**
  If you compile code to run where you know a math co-processor is present (as would always be the case if the APL387.EXE interpreter is running), you should use the option `-on Floating_point` to cause the C compiler to generate co-processor instructions. Two files included with

your package — `hc6.pro` for the APL386 interpreter and `hc7.pro` for the APL387 interpreter — contain options to turn the co-processor on and off.

- **Code for Optional Processor**
  To make the co-processor optional, use the C compiler option `-off Floating_point`. The High-C floating-point emulator requires no special initialization.

  Caution: The NDP FORTRAN compiler must be set to compile for inline floating-point instructions. A machine **must** have a math co-processor capability for APL to call a protected-mode routine compiled with the NDP FORTRAN compiler.

- **Operating System Services from Non-APL Routines**
  You should be able use the Phar Lap DOS Extender Services as long as you do not change the interrupt vectors. See the Phar Lap manuals for details on DOS Extender Services.

- **Stack Segment**
  Your Metaware High-C executable file must include a stack segment. You can include a stack segment in two ways:

  - Specify `pstack.obj` as the first file in the link statement.
  - Use the `-stack` option to the Phar Lap linker. See the sample in the `\demo` directory on your distribution disk.

- **Debugging Protected-Mode Routines**
  You can use the Phar Lap 386|DEBUG or MINIBUG (version 2.2b or later) debuggers to debug protected-mode routines. You must run APL★PLUS II/386 under the debugger. Begin APL with a command like:

```
        386debug apl387.exe
```

  Once APL is running, you can set a breakpoint on an external routine using the *)DEBUG* command.

```
        )DEBUG SUMIT
WAS OFF
```

APL then transfers control to the debugger at the start of the external routine.

```
      SUMIT (⍳3)(3)
Step to 005C:00000004  Elapsed time = 70.8 seconds
EAX=00001887  EBS=00000003  ECX=00000000 EDX=0000003
ESI=0005148C  EDI=003E6A88  EBP=00066D8C ESP=00001FE0
DS=0064  SS=0064  ES=0064  FS=0000  GS=0014
CS:EIP=005C:00000004  EFLAGS=00000346 NV UP EI PL ZR NA PE NC
005C:0004  2BC9                      SUB       ECX,EXX
```

The debugger will not receive keyboard input unless you have the session manager in DOS keyboard mode. Press Alt-F5 to switch in and out of DOS keyboard mode. To clear the breakpoint, use *)DEBUG* again:

```
      )DEBUG SUMIT
    WAS ON
```

Using Real Mode

Calling real-mode routines from APL is more complex and less efficient than calling protected-mode routines. Code generated by a real-mode compiler cannot run from within the memory above the 1024K address line where the APL★PLUS II/386 workspace is located, and it cannot reference addresses in the workspace where its array arguments are located. However, there are many situations that require real-mode routines. For example, protected-mode compilers and linkers are currently specialized and expensive tools. You may already have a conventional compiler for the PC environment or you may want to integrate other packages of real-mode routines into your APL applications.

Version 4 of APL★PLUS II/386 gives you two ways to use real-mode external routines: the original real-mode interface introduced in Version 2, and the externally resident module interface introduced in Version 4. This section describes the original real-mode interface. See the "Using Externally Resident Modules" section in this chapter for details on the new real-mode interface.

Using the Original Real-Mode Interface

The original real-mode interface calls external routines through a language interface program that runs as a resident program. A different interface program is necessary for each language implementation you use. You can install more than one language interface at one time.

When you call a real-mode routine, APL★PLUS II/386 must copy the module and the parameters from extended memory into conventional memory, where the routine runs. APL★PLUS II/386 allocates DOS memory and copies the module into the DOS region. For each parameter, an appropriate area of DOS memory is allocated and the values are copied to that region. APL copies and coerces datatypes of the parameters into the datatypes expected by the routine. A *DOMAIN ERROR* occurs if the parameter cannot be coerced without loss of significance.

APL★PLUS II/386 then places information needed to locate the routine and arguments in a control block in DOS memory. Next, APL★PLUS II/386 issues a software interrupt to a resident language interface program, which sets up the arguments and calls the routine. Figure 12-7 shows the technique.

When the routine finishes, APL★PLUS II/386 copies the module and the items needed for the explicit result back into the workspace and frees all the allocated storage back to DOS.

This release of APL★PLUS II/386 contains the following interfaces for real-mode compilers:

- `aplmsc51.exe` — Microsoft C, versions 5.1 and 6
- `apltbc2.exe` — Borland Turbo C
- `aplfor50.exe` — Microsoft FORTRAN.

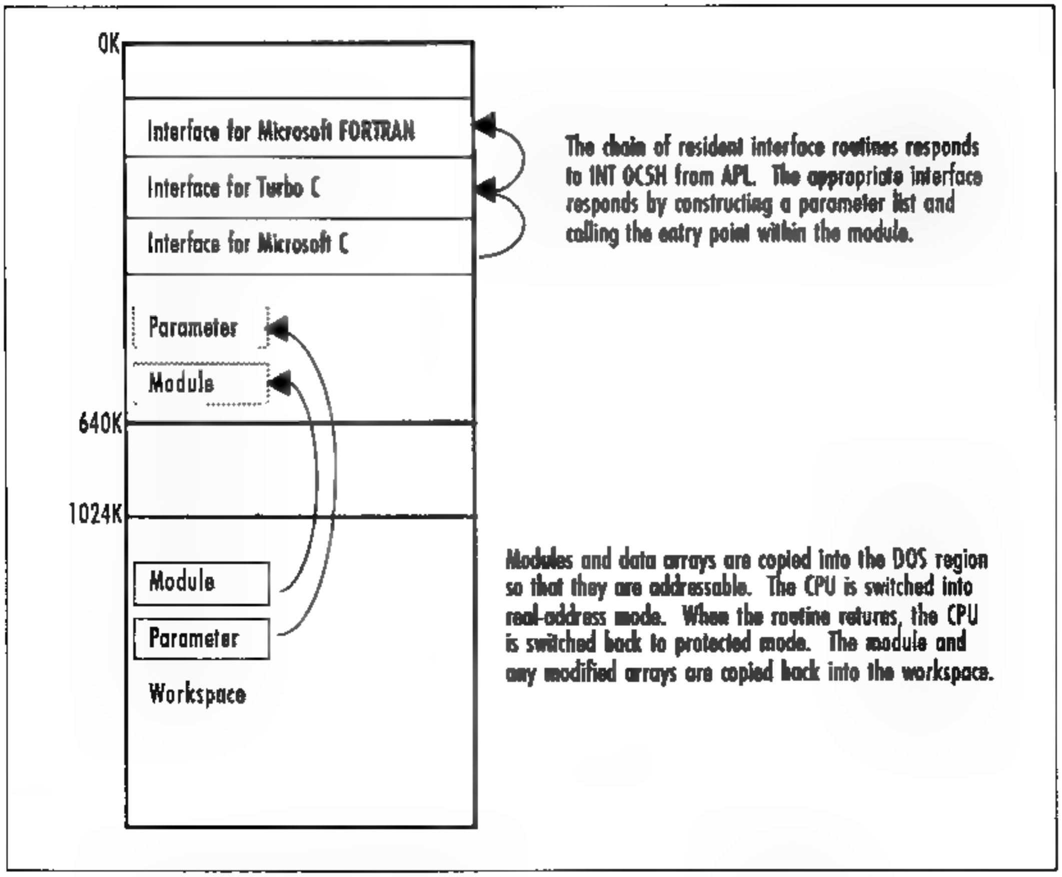

0K
Interface for Microsoft FORTRAN
Interface for Turbo C
Interface for Microsoft C
Parameter
Module
640K
1024K
Module
Parameter
Workspace
The chain of resident interface routines responds to INT 0CSH from APL. The appropriate interface responds by constructing a parameter list and calling the entry point within the module.
Modules and data arrays are copied into the DOS region so that they are addressable. The CPU is switched into real-address mode. When the routine returns, the CPU is switched back to protected mode. The module and any modified arrays are copied back into the workspace.

Figure 12-7. Calling a Real-Mode Routine (Original Interface)

Sample Real-Mode Routine

An example real-mode routine, `sum.c`, is on your distribution disk. This section shows the process of compiling and executing a real-mode routine. This example uses the Borland TURBO C compiler.

The command that compiles the file is:

```
        tcc -c sum
```

The command you use to link the file depends on whether or not the Turbo C floating-point emulator is required. Use the emulator to produce a module that runs with or without a co-processor. If you know that a co-processor is installed, avoid using the emulator to yield faster floating-point calculations.

A sample command to link the file with the emulator is:

```
tlink /m /c tbc2stk \tc\lib\c0s sum,sum6,,\tc\lib\emu
  \tc\lib\maths \tc\lib\cs
```

A sample command to link the file without the emulator is:

```
tlink /m /c tbc2stk \tc\lib\c0s sum,sum7,,\tc\lib\fp87
  \tc\lib\maths \tc\lib\cs
```

Note: You may need to specify a path. The preceding examples include `tbc2stk.obj`, which defines a stack segment for the module and initializes the Turbo C software floating-point emulator, if necessary. (This file is distributed with APL★PLUS II/386.) You **must** include:

- `tbc2stk.obj` as the first object file when you link for Turbo C
- `msc51stk.obj` as the first object file when you link for Microsoft C
- `for50stk.obj` as the first object file when you link for Microsoft FORTRAN.

You must install the language interface program, `apltbc2.exe`, before starting APL:

```
c>apltbc2
APL*PLUS - Turbo C 2.0 Interface, Version 1.1
INT C5 handler installed
```

If you use the floating-point emulator, you should specify the `emulator` option.

```
c>apltbc2 emulator
APL*PLUS - Turbo C 2.0 Interface, Version 1.1
Floating point emulator option
INT C5 handler installed
```

This option is necessary because Turbo C uses different conventions to return an explicit floating-point result with and without the emulator.

Now you can start APL. Load the module with *⎕MLOAD* and create the associated function with *⎕NA*:

```
      SUM ← ⎕MLOAD 'sum.exe,sum.map'
      'TURBOC2 I4←SUM._isum(*I4,I4)' ⎕NA 'ISUMR'
1
      ISUMR (⍳10)(10)
55
```

Technical Specifics

- **Far Calls and Memory Models**
  Real-mode routines are called through an installed language interface program. All interfaces provided by STSC call the external routine with a far call, and pass parameters as far pointers. The module itself can be a mix of memory models, as long as the interface to the routines called from APL uses far calls and far pointers. (See "Technical Specifics" under "Using Protected-Mode Routines" for more details.) Because arrays passed by APL are passed as far pointers, no array passed can be larger than 64 kilobytes.

- **Memory Requirements**
  When you call an associated function, you must have enough DOS storage free for APL to copy the module and arguments into the DOS region. Because space in the DOS region is limited, you may need to adjust several APL startup parameters to ensure that you have enough DOS space left while APL is running. See the Advanced Techniques chapter for a list of techniques.

  Even with careful memory management, you may still not be able to obtain enough space in the DOS region to run large routines. STSC recommends that you use protected mode for running external routines whenever possible.

- **Floating-Point Emulation**
  The real-mode module must be linked in a special way, with an STSC-supplied object file as the first object file. This object file creates a stack segment and performs other initialization. Use `tbc2stk.obj` for Turbo-C; use `msc51stk.obj` for Microsoft C. Control is passed into the module by branching to the first instruction of the module, which will be from this object file. If the module has been linked with the floating-point emulator, this code will initialize the floating-point emulator and then branch into the entry point for the routine being called. If the module has not been linked for floating-point emulation, the emulator initialization is overlaid during the link process.

  Note: Floating-point emulation is not possible for Microsoft FORTRAN routines called from APL. Such routines always require floating-point hardware if floating-point calculations are performed. You must still use `for50stk.obj` to allocate a stack segment.

- **File Operations**
  You should be able to use file operations as long they do not change the DOS, APL, or the 386 | DOS Extender environments.

- **Other Environment Initialization**
  If you need to perform other initialization actions on the call to every routine within a module, add them by modifying the distributed assembler file and re-assembling it. You can also modify this file to adjust the stack size for the module.

- **Debugging Real-Mode Routines**
  You can use a real-mode debugger (such as DEBUG or SYMDEB) to debug real-mode routines as they are called from APL. Run the session manager under the debugger; for example:

  ```
          debug apl387.exe
  ```

  Once APL is running, you can set a breakpoint on the routine using *)DEBUG*. See "Debugging Protected-Mode Routines," earlier in this chapter for more information on *)DEBUG*.

Details on the Language Interface

This section explains the structure of a language interface program and how it invokes an external routine and returns the results to APL. The source code for all of the sample interface programs is provided on your distribution disks so you can see how they work.

What the Language Interface Does

The language interface program is a terminate-and-stay-resident program. You can load into memory as many interface programs as you need before starting APL.

When you call an external routine, the address of the entry point and information on its parameters are placed in a small control block, whose address is passed in a register when APL issues the Int C5h to call the interface routine.

The role of the interface program is to use the data in the control block to set up arguments in the manner appropriate to the module's language and compiler. Using this information, the language interface program finds the module and arranges the parameters as the external routine expects them, then calls the external routine.

If you have more than one language interface program loaded, they form a "chain" in memory. Every interface knows the name of the language it is designed to handle. You specify this language name, such as MSC51 or TURBOC2, when you create the associated function with *⎕NA*. When an associated function calls a real-mode routine, APL passes this language name in the Int C5h call. If this matches the interface program's language name, that program handles the call. If not, it passes the language name to the next interface, until the correct interface is found. If you did not install the interface, APL★PLUS II/386 issues the error *LANGUAGE INTERFACE NOT FOUND*.

To remove a language interface, use the `unhook.com` command found on your distribution disks. This command removes the most recently installed language interface.

Writing Your Own Language Interface Programs

If you are an experienced programmer, you can write your own language interface programs.

Caution: You should not attempt to write interfaces unless you are experienced with 8086 assembler and low-level programming in the DOS environment.

The source for the Turbo C interface is in the file `apltbc2.asm` on the distribution disk. Use it as a guide to writing your own language interface. It includes information on how Int C5h handlers are chained, how a language interface recognizes a call that it should process, and how the module and parameters are passed from APL.

Contact STSC's Professional Services group if you would like STSC to develop a custom interface.

Using Externally Resident Modules

This version of APL★PLUS II/386 features a new version of the real-mode interface that allows you to call external routines without bringing them into the workspace. This section explains why this interface was developed, describes how the interface works, and describes the sample interface provided with this version of APL★PLUS II/386.

The original real-mode interface of ⎕*NA* cannot take advantage of many compiler and third-party library routines, such as database interfaces, that you call from C routines. This limitation is inherent in the interface because it cannot initialize the C run-time environment necessary to support these libraries. For example, because the original interface cannot set memory allocation heap pointers and other global variables, you cannot use C runtime library routines such as `malloc` from APL★PLUS II/386.

Another limitation of the original interface is the per-call overhead you incur with external routines packaged into large modules. Each time you call such an external routine, APL★PLUS II/386 copies the entire module from protected mode to real mode, runs the routine, and then copies the module back to protected mode. This overhead is particularly problematic when you call relatively simple functions with small arguments, since it can significantly degrade performance.

Lastly, a less-common problem occurs because the module is not in a fixed location. Because the module resides in the APL workspace, it is subject to movement as part of normal APL workspace memory management. This means that you cannot interact with software that relies on fixed buffer locations within your module, which is a common requirement for database interfaces and other packages.

To address these limitations, this version of APL★PLUS II/386 includes a technique for initializing and calling externally resident modules from APL★PLUS II/386. Using this technique allows you to avoid the limitations formerly associated with calling real-mode routines.

What Is An Externally Resident Module?

An externally resident module is a suite of external routines that you have compiled and linked into an executable file. This file also contains information that initializes a special language interface and calls APL★PLUS II/386.

An externally resident module has four primary parts: `main()` routine, a callback table, a collection of executable routines, and the APLMC60X.ASM language interface module.

- **`main()` Routine**
  The `main()` routine is a traditional C language `main()` routine. This routine initializes the C language run-time support and calls the `AplLink` function. The `AplLink` function activates the language interface and then starts APL★PLUS II/386.

- **Callback Table**
  A callback file lists the names of the programs that you want to call from APL★PLUS II/386. The file is compiled into a callback table, which is linked with external module. The language interface uses the callback table to locate a particular routine.

- **Callable Routines**
  The next primary part is the collection of callable routines that you want to call from APL★PLUS II/386. These can be routines that you write or library routines from another software package.

- **APLMC60X.ASM**
  This is an assembler module that you assemble and line into the externally resident module. It contains the logic needed to map a call from APL★PLUS II/386 to the callable routines in the externally resident module.

You package an externally resident module as an .EXE file that you call directly from DOS. It performs any necessary initialization and then starts APL★PLUS II/386. The initial workspace that you specify for APL★PLUS II/386 typically has a latent expression that uses ⎕*NA* to create associated functions for the callable routines in the external module.

With externally resident modules, the bulk of the code resides completely outside of the APL workspace in a fixed location. It does not move between protected mode and real mode each time you call an external routine within the module. Instead, you use ⎕*MLOAD* to load a very small "stub" module into the APL workspace. This stub module contains small surrogate functions that correspond to each routine you want to call in the externally resident module. These surrogates are not executable functions; they contain only function linkage information that describes the actual routines.

APL★PLUS II/386 treats the stub module as a regular real-mode module and copies it between protected mode and real mode on each call. However, because the stub module is so small, moving it does not introduce substantial overhead.

You then use ⎕*NA* to define an associated function for each external routine that you want to call in the externally resident module. When you run that associated function, the linkage logic in APLMC60X.ASM uses the stub module to look up the location of the exernal routine in the externally resident module and calls it with the parameters from APL★PLUS II/386. The result is passed back and becomes the explicit result of the associated function.

There is not a significant overhead associated with the table lookup operation. Once you run a routine, APLMC60X.ASM "remembers" where the routine is located in the callback table and can go directly to the routine the next time you call it.

Example of Using ⎕*NA* with Externally Resident Modules

The following steps show an example of how you use ⎕*NA* with externally resident modules. The examples in these steps use the XCODE group of files supplied with APL★PLUS II/386.

1. Initialize the interface between APL★PLUS II/386 and the externally resident module This example uses the XCODE.EXE module, calls APL387.EXE, and uses a configuration file named CONFIG APL.

    ```
    xcode apl387 config=config.apl
    ```

    Before XCODE.EXE starts APL★PLUS II/386, it installs itself as a language interface module by establishing a handler for interrupt 0C5h, analogous to the language interfaces you install when you use the original interface.

2. Use ⎕*MLOAD* to load the stub module into APL★PLUS II/386. This stub module, XSTUB.EXE, contains the descriptors that locate the entry points in XCODE.EXE.

    ```
          MYXFNS←⎕MLOAD 'C:\APLII\XSTUB.EXE'
    ```

3. Use ⎕*NA* to create an associated function to call each routine you want to use. This example uses the *X*0 datatype to pass a nested array argument and return the result, and uses the disclose (⊃) and enclose (⊂) prefixes to enclose the argument passed by the associated function and disclose the result returned by the external routine.

    ```
      'XCODE ⊃X0← MYXFNS._Xpack(⊂X0)' ⎕NA 'Xpack'
    1
    ```

    The result indicates that the definition is successful.

4. Run the associated function. The associated function transfers the arguments to the external routine.

    ```
          Xpack ⍳3
    32 2 16908376 3 3 1 2 3 36
    ```

5. The external routine runs and generates its result. The result is transferred back to the workspace.

6. When you leave APL★PLUS II/386, the externally resident module exits to DOS.

Technical Specifics of Externally Resident Modules

This section gives more detail on the `main()` routine, callback file, and calling multiple externally resident modules.

- **`main()` Routine**
  The STSC-supplied `main()` routine contains the `AplLink` function that installs the language interface and starts APL★PLUS II/386. You can use this routine, modify it, or use your own. The default `main()` routines collects all the command line arguments and passes them to the `AplLink` function as the command line to start APL★PLUS II/386. Your routine can interpret some or all of the command line arguments for its own purpose. If your `main()` routine does this, do not pass these arguments to `AplLink` as part of the command that starts APL★PLUS II/386.

  `AplLink` uses whatever command you pass it to start APL★PLUS II/386. The first name of the command line that you pass must specify the name of a .EXE or .COM file, typically the full name of the APL interpreter. If you do not specify an explicit path, `AplLink` searches your current directory for a .COM file, then for a .EXE file. It repeats this process for each directory specified by the PATH environment variable. `AplLink` does not search for or execute .BAT files.

  If you call `AplLink` with an empty command, it looks for the environment variable APLCMD and uses that value as the APL command line.

- **Callback File**
  The callback file consists of the names of the external routines you want to call from APL★PLUS II/386. Each entry uses the APLCB macro. You construct this file as a file with a .CB extension. You compile this file into both the externally resident module and the stub module by using a `#include` statement in your C source program. Each entry in the callback file takes the form

  ```
      APLCB( functionname )
  ```

  where *functionname* is the name of the external routine you want to call. A typical callback file might include entries like:

  ```
      APLCB( MyFunction )
      APLCB( YourFunction )
      APLCB( TheirFuntion )
  ```

  Note that these entries do not end in a semicolon. The APLCB macro generates a different type of entry depending on the module that you compile it into: the stub module or the externally resident module.

  When compiled into the stub module, APLCB generates a linkage stub. The linkage stub is a data structure that contains the routine name and internal fields used to optimize access to the routine. This optimization means that the interface only needs to look up the routine once.

  When compiled into the externally resident module, APLCB generates an entry in the callback table. This table establishes the correspondence between the routine names in the stub module and the actual routine in the externally resident module.

- **Calling Multiple Externally Resident Modules**
  If you must install multiple externally resident modules for your application to run, you can "chain" the modules together on the DOS command line. To use this technique, your application must pass its command line arguments to `AplLink` in a manner similar to the example code distributed

with APL★PLUS II/386. For example, if you have three externally resident modules that you need to install, you can use the following line to call the modules and start APL★PLUS II/386:

```
xmod1 xmod2 xmod3 apl387 config=config.apl ws=mine
```

Using Templates to Build or Modify Exernally Resident Modules

STSC provides a set of "template" files that you can use to build your own externally resident modules. These templates are in the \REAL\MSC directory on the Interfaces distribution disk. You may want to study these examples in addition to the XCODE examples as you read this section. The make files in this directory are a good place to begin, because then show how all of the pieces fit together to make the externally resident module and its matching stub module.

You can use the functions in the *TMPLCOPY* workspace to copy and rename the template files so that they have the appropriate names for your application. The four template files are:

- TMPLCODE.C — the C source code for the externally resident module. This file contains the `main()` routine and defines the `AplLink` function.
- TMPLSTUB.C — the C source code for the stub module. You should not have to modify this file.
- TMPLCODE.CB — the callback file for the externally resident module.
- TMPLCODE.MAK — the make file you use with the Microsoft C compiler to generate the externally resident module and stub module.

Load the workspace and run the function *TMPCOPY*. *TMPLCOPY* has the syntax:

```
['directory'] TMPLCOPY 'codename' 'stubname'
```

where:

directory	The optional directory name enclosed in quotes that contains the template files. This directory also specifies where you create the new module and stub files. The default is your current directory.
codename	The name of the source code file for the externally resident module. Enclose this name in quotes as the first item of a nested vector.
stubname	The name of the source code file for the stub module. Enclose this name in quotes as the second item of a nested vector.

For example, to generate the files necessary to build an externally resident module named MYCODE with a stub module name MYSTUB, you enter:

```
      TMPLCOPY 'MYCODE' 'MYSTUB'
```

TMPLCOPY copies and renames the four template files, so that you now have:

- MYCODE.C
- MYSTUB.C
- MYCODE.CB
- MYCODE.MAK.

Once you have the template files, you can follow these steps to build your externally resident module and stub module. These steps use the file names shown in the above example.

1. Edit the newly created MYCODE.CB file. Add an APLCB entry for each routine you want to call from APL★PLUS II/386; for example:

   ```
   APLCB( Triplets )
   APLCB( Xpack )
   APLCB( Xunpack )
   ```

2. Edit the newly created MYCODE.C file. You can define your external routines in this file in three ways:

   - You can define them directly in the file, as indicated by the file comments.
   - You can define them in another module — you must use the `#include` command to include a header file with the function prototypes for the routines mentioned in the callback file.
   - You can access them from predefined libraries — you must use the `#include` command to include the appropriate header file.

   The XCODE sample code supplied with APL★PLUS II/386 uses the second method. It uses the header file XFNS.H to reference the routines defined in the XFNS.C file.

3. Edit the newly created MYCODE.MAK file to include appropriate instructions for compiling all of the components of your application. The template file contains comments that can guide you.

Using the ⎕*NA* Tools

This section describes three routines that illustrate techniques for manipulating nested arrays and using the ⎕*NA* tools. To follow along with the examples, start APL★PLUS II/386 with the XCODE externally resident module; for example:

```
xcode apl387 config=c:\aplii\config.apl
```

XCODE installs the appropriate language interface and calls APL★PLUS II/386. The associated functions used as examples in this section are in the *XFNS*, *XCODE*, and *PXFNS* workspaces. If you started APL★PLUS II/386 with the XCODE externally resident module, use the *XCODE* workspace to follow along with the examples. This workspace contains the example functions discussed in this section. It also contains the *Bind* function, which performs the ⎕*MLOAD* and ⎕*NA* operations you need to do to define associated functions.

The XFNS.C sample file illustrates the use of the *X0* datatype to analyze and construct APL vlaues for an external routine. XFNS.C contains three sample routines: `Xpack`, `Xunpack`, and `Triplets`.

The `Xpack` routine takes an arbitrary APL array of any type, shape, or depth and packages its internal representation within an integer vector shell. The associated function for the `Xpack` routine is in the *XFNS*, *XCODE*, and *PXFNS* sample workspaces; it is also called *Xpack*. You call *Xpack* as shown below:

```
      P←Xpack ⍳9
      P
60 2 16908376 9 9 1 2 3 4 5 6 7 8 9 60
```

You can write this value to a native file or into a database. You can convert (or "unpack") it back to its original value with the *Xunpack* function; for example:

```
      Xunpack P
1 2 3 4 5 6 7 8 9
```

The C source code for these functions is shown below. They are relatively simple, yet powerful, routines.

`Xpack` locates the overall length count for the argument array. It then determines how many integer elements (four bytes each) it needs to "contain" the argument. The routine then calls the `avMakeIntVector` routine to allocate an APL array and fill it with the packed argument's value. To keep the original argument's values packed, the routine does not call `avUnpack` for this argument. However, the routine does call `avPack` for the newly created integer result before APL★PLUS II/386 accepts it.

Because this is not a nested array, packing is not really necessary; however, values returned to APL★PLUS II/386 must have a "validation stamp" of X and 0 in the ninth and tenth bytes. The `avPack` routine handles this. The values received from APL★PLUS II/386 are also "stamped" in this way. This is why it is not necessary to call `avPack` to prepare the result of `Xunpack`. In fact, such a call is disastrous if the values returned are nested or heterogeneous (`PTR` or `HET` descriptor types), because `avPack` assumes that the pointers within the argument are absolute addresses. The pointers are actually relative displacements from the beginning of the packed value.

The `Triplets` routine found in XFNS.C and the *XFNS*, *XCODE*, and *PXFNS* workspaces is complex. This section does not describe it in detail, but illustrates its behavior in APL★PLUS II/386. This routine is useful only as an illustration of complex nested array programming techniques; it is not meant as a production routine. These examples use the *DISPLAY* function to illustrate the results; *DISPLAY* is included in the workspaces.

The following example shows how this routine works.

```
      DISPLAY Triplets (1 2) (3 4) ('AB' 'CD')
.→-------------------------------.
| .→----------.  .→----------.   |
| |      .→-. |  |      .→-. |   |
| | 1 3  |AB| |  | 2 4  |CD| |   |
| |      '--' |  |      '--' |   |
| '∊----------'  '∊----------'   |
'∊-------------------------------'
```

This is essentially the same as

```
      DISPLAY ↓[1]↑(1 2) (3 4) ('AB' 'CD')
.→--------------------------.
| .→---------- .→--------- . |
| |       .→-. | |       .→-. | |
| | 1 3 |AB| | | 2 4 |CD| | |
| |       '--' | |       '--' | |
| '∊---------' '∊---------' |
'∊--------------------------'
```

but *Triplets* also does scalar extension:

```
      DISPLAY Triplets 1 (2 3 4) ('AB' 'CD' 'EF')
.→-----------------------------------------.
| .→---------- .→---------. .→---------. |
| |       .→-. | |       .→-. | |       .→-. | |
| | 1 2 |AB| | | 1 3 |CD| | | 1 4 |EF| | |
| |       '--' | |       '--' | |       '--' | |
| '∊---------' '∊---------' '∊---------' |
'∊-----------------------------------------'
```

The scalar (1) is coupled with 1, 2, 3 and 'AB', 'CD', 'EF' respectively.

Triplets does another kind of scalar extension. The third item can be either a nested vector of character vectors or a simple character vector (scalars are treated as vectors). The previous example shows what happens with nested vectors of vectors. With simple character vectors, *Triplets* does an implicit enclose; that is, it treats the argument 'ABCD' as if you specified it as (⊂'ABCD). The next example shows this implicit enclose.

```
      DISPLAY Triplets (1 2) (3 4) 'FIVE TWICE'
.→-------------------------------------------.
|.→-----------------.  .→-----------------.|
||      .→---------. |  |      .→---------. ||
|| 1 3 |FIVE TWICE| |  | 2 4 |FIVE TWICE| ||
||      '----------' |  |      '----------' ||
|'∊------------------'  '∊------------------'|
'∊-------------------------------------------'
```

If you omit the last argument, *Triplets* implicitly generates a nested scalar by enclosing an empty character vector; that is, ⊂'', as shown in the next example.

```
      DISPLAY Triplets (⍳3) (3?100)
.→------------------------------------------.
| .→---------. .→---------. .→---------.    |
| |      .⊖. | |      .⊖. | |      .⊖. |    |
| | 1 87 | | | | 2 25 | | | | 3 54 | | |    |
| |      '-' | |      '-' | |      '-' |    |
| '∊---------' '∊---------' '∊---------'    |
'∊------------------------------------------'
```

Last, *Triplets* accepts a matrix argument. It uses this argument by implicitly splitting it along the first axis; for example:

```
      X←(3 2⍴⍳6),'AB' 'CD' 'EF'
      DISPLAY Triplets X
.→------------------------------------------.
| .→---------. .→---------. .→---------.    |
| |     .→-. | |     .→-. | |     .→-. |    |
| | 1 2 |AB| | | 3 4 |CD| | | 5 6 |EF| |    |
| |     '--' | |     '--' | |     '--' |    |
| '∊---------' '∊---------' '∊---------'    |
'∊------------------------------------------'
```

So far, this section has shown how this function behaves externally. To see how the routine is constructed in C, you can examine the source code found in XFNS.C. If you have a real-mode debugger, such as Codeview, you can use it to see, step-by-step, what happens within the routine. To do this, you must use Codeview with the externally resident sample code in XCODE.EXE. Call XCODE.EXE under Codeview.

cv xcode *aplcommandline*

where *aplcommandline* is the normal APL★PLUS II/386 command line. The source file is divided into five main sections:

- Section 1
  Begins with the `avUnpack` routine and ends at the "Split argument . . ." comment. Section 1 performs basic initialization. It unpacks the argument, checks its rank and element count, and initializes the key variables. The set of "`av`" prefix variables are important to initialize here as NULL. The routine uses these variables in the last section (Section 5) to free these temporary values. The `avFree`

routine behaves properly when passed a NULL pointer, but the routine fails if it reaches Section 5 without the three values (`avH`, `avD`, and `avT`) having an APL value pointer or being NULL.

- Section 2
  Begins with the "Split argument . . ." comment and ends at the "Build result shell" comment. Section 2 is where most of the work occurs. First, the routine sets up three `AplVal` pointers corresponding to the three argument items or columns as `avH`, `avD`, and `avT`. For vector arguments, the routine uses `avFetchPTR` to do this. For matrix columns, `Triplets` increments between items to access the values "in place" within the original array. For the first column, the routine sets `avH` equal to the argument `avIn` (but increment its reference count with `avIncRefCount`), since it shares the same value. Then, the routine sets `itemH` = 0 and `incrH` as 2 or 3, depending on how many columns are in the array. For columns 2 and 3, the routine sets `itemD` and `itemT` as 1 and 2, respectively. When the routine actually accesses the array (in Section 4), the routine adds the increment to the item to get the next item. Section 2 is also where the routine "measures" the argument sizes in preparation for Section 3.

- Section 3
  Begins with the "Build result shell" comments and ends at the "Restructure each subitem . . ." comment. In Section 3, the result "shell" is allocated as a `HET` array. The routine allocates an appropriate prototype if the array is empty.

- Section 4
  Begins with the "Restructure each subitem . . ." comment and ends at the "Cleanup and exit" comment. In Section 4, the routine actually "fetches" items from the arguments and builds the "triplet" arrays that make up the result.

- Section 5
  Begins with the "Cleanup and exit" comment and ends at the next comment. The next comment begins the definition of the `Xpack` routine. Section 5 frees the temporary arrays used in sections 2 and 4. It also contains all the error signaling logic. Note how carefully the routine frees values that are not

used again. You do not have to be so careful with your own code, because any values allocated with the functions in APLVAL.H and APLEXT.H have their storage released automatically when the external routine returns to APL★PLUS II/386.

Tools for Building and Modifying External Routines

This section describes the suite of routines you can use to build and modify your own external routines. These tools are all in the `\real\msc` and `\prot\mwc` subdirectories on the Interfaces disk.

The external routine tools and sample external routines are distributed as C source code. You must compile and link them to use them.

How you install and maintain these tools depends on the number, extent, and complexity of your external routines. Although you can change the tools to adapt them to your particular needs, you should maintain an unmodified reference copy of the source code as it is distributed.

If you only want to run the sample external routines, you can use the Microsoft C NMAKE program (or a similar utility) to process the file. Examine the files XFNS.MAK, PXFNS.MAK, and XCODE.MAK to see what they do and to ensure that path names are appropriate for your setup.

If you are developing complex applications or modifying the support functions, you may want to incorporate the source files into your application's source maintainance scheme. If your site has several external routines, you may want to build a library archive containing precompiled versions of the tools, and incorporate the header files in a standard place.

The source code for the tool functions contains many instances of the `assert()` macro. You should consider these occurrences an essential part of the internal documentation of the routines and their parameters. By default, an `assert()` macro call expands into code that tests the truth of its argument expression. This code prints an error message and stops the process if the expression is false. You may want to disable these tests in the production version of a thoroughly tested process designed to completely check the arguments received from APL. See the description of the `assert()` macro in your C language documentation.

Note: If you use these tools from protected mode or with the original real-mode interface, you should load the module with *⎕MLOAD* each time you run your application. Doing this initializes the memory pointers properly.

The header file APLEXT.H contains data structures and macros you can use to build APL★PLUS II/386 external routines. In particular, it defines three important routines:

- `AplAlloc` allocates a block of temporary DOS memory that is automatically freed on return to APL.
- `AplFree` explicitly frees a block of temporary DOS memory allocated by `AplAlloc()`.
- `AplError` signals an error in APL★PLUS II/386.

AplAlloc

Purpose:	Use `AplAlloc` to allocate a block of memory.
Syntax:	`void far * AplAlloc(` *size* `)` `unsigned long` *size*
	size Specifies the size of the block of memory in bytes
Effect:	Allocates a block of memory of the specified size. If the routine succeeds, it returns a pointer to the allocated memory location. If the routine fails, it returns a NULL. This memory is

temporary; it is freed when you explicitly call `AplFree` or as soon as control returns to APL★PLUS II/386.

`AplFree`

Purpose: Use `AplFree` to explicity free a block of memory that you allocated with `AplAlloc`.

Syntax:

```
AplFree( pointer )
void far * pointer
```

pointer Specifies the pointer that is returned by `AplAlloc`.

Effect: Frees the block of memory that was allocated with `AplAlloc`.

`AplError`

Purpose: Use `AplError` to signal an error to APL.

Syntax:

```
AplVal AplError( message )
char FAR * message;

extern int aplerror;
```

message Specifies the text of the error message or a NULL.

aplerror An external variable set to 1 if `AplError` is called with a non-NULL argument. If you want to test it to determine when an error has occurred in a subroutine, you must manually reset this value to 0 on initial entry to the external routine.

Effect: Signals an error to APL★PLUS II/386. The error does not take effect until the external routine returns to APL. Use a NULL argument to cancel an error that `AplError` previously signaled.

Only the most recent call to `AplError` prior to return to APL takes effect. As a convenience, `AplError` returns a NULL pointer so that you can use it in combination with the "return" statement; for example:

```
return AplError( "DOMAIN ERROR" );
```

This routine also sets the external variable `aplerror` to 1 if you call it with a non-NULL argument.

Basic Macros for Manipulating APL Values

The header file APLVAL H contains C language macros for accessing the data structures that represent APL values. These macros are similar to the macros described in the VARINFO.TXT file distributed with APL★PLUS II/386. They constitute the lowest-level definition of the data structures shared with the APL interpreter. Most of these macros evaluate to a pointer and not a value. Their use is shown in the source code for the higher-level functions described in the rest of this section. Avoid using them in new code unless necessary; instead, use the functions described in the section, "Basic Functions for Manipulating APL Values."

Basic Functions for Manipulating APL Values

The functions described in this section are more convenient for manipulating APL values than the macros described in the preceding section.

The necessary declarations for these routines are in the header file APLVAL.H. Your external routine should contain the following line:

```
#include "aplval.h"
```

APL values are represented by the opaque type `AplVal`. An `AplVal` is a pointer to a data structure that contains both the value's data itself and information about its shape, representation, and so forth. The details of an `AplVal`'s internal structure are hidden in the routines described in the following sections. Using these routines, you can write programs that work on different machine architectures and that you can readily update if the internal form of variables changes in future versions.

Note: Microsoft Windows uses different naming conventions than DOS; therefore, the functions available for use under Windows have slightly different names that conform to those conventions. The naming conventions used here are compatible with those used in the Using External Processes chapter in the *APL★PLUS II for UNIX User Manual* (STSC, 1991). Using these conventions can make it easier to port routines between the DOS and UNIX environments, although other important changes are necessary. If you prefer to use the Windows naming conventions, you can use the routines described in the *APL★PLUS II/386 Windows Interface* manual.

Table 12-3 summarizes the available functions.

Table 12-3. **Basic Functions for Manipulating APL Values**

Function	Description
`avDesc`	Obtain the representation type of an APL value.
`avPDesc`	Obtain a pointer to the representation type of an APL value.
`avNelm`	Obtain the number of elements in an APL value.
`avPNelm`	Obtain a pointer to the number of elements in an APL value.
`avRefCount`	Obtain the reference count of an APL value.
`avIncRefCount`	Increment the reference count of an APL value.
`avDecRefCount`	Decrement the reference count of an APL value.
`avPShape`	Obtain a pointer to the shape vector of an APL value.
`avRank`	Obtain the rank of an APL value.
`avPRank`	Obtain a pointer to the rank of an APL value.
`avPValuesINT`	Obtain a pointer to the elements of an APL value with `INT` representation.
`avPValuesFLOAT`	Obtain a pointer to the elements of an APL value with `FLOAT` representation.
`avPValuesCHAR`	Obtain a pointer to the elements of APL value with `CHAR` representation.
`avPValuesBOOL`	Obtain a pointer to the elements of an APL value with `BOOL` representation.
`avPValuesPTR`	Obtain a pointer to the elements of an APL value with `PTR` representation.
`avAllocate`	Allocate storage and construct a new APL value.
`avFree`	Release the storage used by an APL value created by `avAllocate`.

Table 12-3. continued

Function	Description
`avFetchHetDesc`	Access the descriptor of an item in a heterogeneous APL value.
`avFetchHetCHAR`	Access the value of a character item in a heterogeneous APL value.
`avFetchHetBOOL`	Access the value of a Boolean item in a heterogeneous APL value.
`avFetchHetINT`	Access the value of an integer item in a heterogeneous APL value.
`avFetchHetFLOAT`	Access the value of a floating-point item in a heterogeneous APL value.
`avFetchHetPTR`	Access the value of a pointer item in a heterogeneous APL value.
`avFetchPTR`	Access the value of a pointer item in any type of APL value.
`avSetHetCHAR`	Assign a character value to an item in a heterogeneous APL value.
`avSetHetBOOL`	Assign a Boolean value to an item in a heterogeneous APL value.
`avSetHetINT`	Assign an integer value to an item in a heterogeneous APL value
`avSetHetFLOAT`	Assign a floating-point value to an item in a heterogeneous APL value
`avSetHetPTR`	Assign a pointer value to an item in a heterogeneous APL value.
`avSetPTR`	Assign a pointer value to an item in a heterogeneous APL value and perform `avFree` of any `PTR` value previously defined for the item being set.
`avHetBoolToInt`	Coerce all top-level Boolean values in a heterogeneous value to integer representation.
`avMakeCharVector`	Construct an APL character vector value from a null-terminated string.
`avMakeIntVector`	Construct an APL integer vector.
`avMakeFloatVector`	Construct an APL floating-point vector.

Table 12-3. continued

Function	Description
`avMakeIntScalar`	Construct an APL integer scalar from a C `long`.
`avMakeFloatScalar`	Construct an APL floating-point scalar from a C long.
`avMakeCharScalar`	Construct an APL character scalar from a C `long`.
`avFetchLong`	Fetch and coerce an integer value from an APL value.
`avFetchBOOL`	Fetch and coerce a Boolean value from an APL value.
`avSetBOOL`	Assign to an element of an APL value with `BOOL` representation type.
`avCoerce`	Coerce an APL value from one type to another.
`avAllocEmptyMat`	Create an empty integer or character matrix.
`avAllocEmptyVec`	Create an empty integer vector.
`avAllocEmptyCharVec`	Create an empty character vector.
`avPack`	Packs an *X*0 value being returned as an explicit result to APL.
`avUnpack`	Unpacks an *X*0 argument. This argument must be unpacked before you can access it with other data access routines.

You can use these routines in real-mode or protected-mode functions. However, because of the segmented architecture of DOS machines, you must be concerned about the differences between near pointers and far pointers. APL values are represented using 4-byte pointers. Under real mode, this representation means that you use far (16:16) pointers to the data, which place a limit of 2*16 bytes on APL arrays in real mode. In protected mode, you must use near pointers, which means that only the .REX model is appropriate for manipulation of X0 data in protected mode. You cannot use the .EXP model because it uses far (16:32) pointers, which do not fit within the 32-bit pointer size used by the arrays.

The APLVAL.H header file automatically generates the appropriate near or far pointers for the Microsoft C and Metware High C compilers. You may need to explicitly define FAR=far or FAR=near in your makefile for other compilers.

The Representation Type

The way an APL value stores data depends on the range of values for the array's elements. The representation type descriptor of a particular value encodes how that value stores data. An external routine generally handles values of each type differently.

The possible types are:

- `CHAR`
  Character data; one byte per element.
- `BOOL`
  Boolean data; eight elements per byte.
- `INT`
  Integer data; four bytes per element.
- `FLOAT`
  Floating-point data; eight bytes per element.
- `PTR`
  Nested data; four-byte pointer to another APL array.
- `HET`.
  Heterogeneous data; 10-byte elements with individual descriptors and space for all of the preceding types.

`CHAR`, `BOOL`, `INT`, `FLOAT`, `PTR`, and `HET` are small integer constants defined by the APLVAL.H header file.

avDesc

Purpose: Use `avDesc` to obtain the representation type of an APL value.

Syntax:

```
int avDesc( apl_value )
AplVal apl_value
```

apl_value	Specifies an APL value.

Effect: The representation type of *apl_value* is returned. This type is a small integer equal to one of `CHAR`, `BOOL`, `INT`, `FLOAT`, `PTR`, or `HET`.

avPDesc

Purpose: Use `avPDesc` to obtain a pointer to the representation type field of an APL value.

Syntax:

```
unsigned char * avPDesc( apl_value )
AplVal apl_value;
```

apl_value	Specifies an APL value.

Effect: This returns a pointer to the field containing the representation type of *apl_value*. It should not be necessary to use this function, since this field is initialized by the routines that create APL values.

The Number of Elements

Each APL value contains a field that records the number of elements in the represented array. This number is always equal to the product of the elements of the array's shape vector.

avNelm

Purpose: Use `avNelm` to obtain the number of elements in an APL value.

Syntax: `long avNelm(` *apl_value* `)`
`AplVal` *apl_value;*

apl_value Specifies an APL value.

Effect: The total number of elements in the value is returned. For a properly constructed `AplVal`, this number will be equal to the product of the elements of its shape.

avPNelm

Purpose: Use `avPNelm` to obtain a pointer to the number of elements in an APL value.

Syntax: `long * avPNelm(` *apl_value* `)`
`AplVal` *apl value;*

apl value Specifies an APL value.

Effect: This returns a pointer to the field containing the number of elements in *apl_value*. It should not be necessary to use this function, since this field is initialized by the routines that create APL values.

The Reference Count

Each APL value contains a reference count field. The reference count should be maintained equal to the number of permanent pointers that refer to the value.

avRefCount

Purpose: Use `avRefCount` to obtain an APL value's reference count.

Syntax: `long avRefCount(` *apl_value* `)`
`AplVal` *apl_value;*

apl_value Specifies an APL value.

Effect: The value's reference count is extracted and returned.

avIncRefCount

Purpose: Use `avIncRefCount` to increment the reference count.

Syntax: `AplVal avIncRefCount(` *apl_value* `)`
`AplVal` *apl_value;*

apl_value Specifies an APL value.

Effect: The value's reference count increases by one. The argument value is returned in the result as a convenience, so that the reference count of an item can be incremented while passing the value as an argument to another function.

avDecRefCount

Purpose: Use `avDecRefCount` to decrement the reference count.

Syntax: `void avDecRefCount(` *apl_value* `)`
`AplVal` *apl_value;*

apl_value Specifies an APL value.

Effect: The value's reference count decreases by one.

The Shape Vector

Each APL value contains a shape vector that describes the dimensions of the array represented. The value's rank gives the number of elements in the shape vector: 0 for a scalar, 1 for a vector, 2 for a matrix, and so forth. The elements of the shape vector must be non-negative. The product of the elements must be equal to the number of elements returned by `avNelm`.

avPShape

Purpose: Use `avPShape` to obtain a pointer to an APL value's shape vector.

Syntax: `long * avPShape(` *apl_value* `)`
`AplVal` *apl_value;*

apl_value Specifies an APL value.

Effect: You can use statements such as

```
rows = avPShape( someMatrix )[ 0 ];
columns = avPShape( someMatrix )[ 1 ];
avPShape( someVector )[ 0 ] = rows * columns;
```

to access and set the elements of a shape vector. The subscripts must be non-negative and less than the APL value's rank.

The Rank

The rank of an APL value is the number of elements in its shape vector. In most cases, the rank is established when the APL value is created and never changed afterwards. The highest rank possible in APL★PLUS II/386 is 127.

avRank

Purpose: Use `avRank` to obtain the rank of an APL value.

Syntax:

```
int avRank( apl_value )
AplVal apl_value;
```

apl_value	Specifies an APL value.

Effect: This returns *apl_value*'s rank. The rank is never negative. The rank of a scalar is 0.

avPRank

Purpose: Use `avPRank` to obtain a pointer to the rank of an APL value.

Syntax:

```
unsigned char * avPRank( apl_value )
AplVal apl_value;
```

apl_value	Specifies an APL value.

Effect: This returns a pointer to the field containing *apl_value*'s rank. It should seldom be necessary to use this function, since this field is initialized by the routines that create APL values.

The Array Elements

The elements of an APL array are stored in ravel order within an APL value. The functions in this section return pointers to the first element. To access and set elements of APL values, use C expressions such as:

```
a_long = ( avPValuesINT( some_INT_vector ) )[j];
a_double = ( avPValuesFLOAT( some_FLOAT_vector ) )[j];
a_char = ( avPValuesCHAR( some_CHAR_vector ) )[j];

(avPValuesINT( some_INT_vector ) )[j]=some_long;
(avPValuesFLOAT( some_FLOAT_vector ) )[j]=some double;
```

avPValuesINT

Purpose: Use `avPValuesINT` to obtain a pointer to the elements of an APL value with `INT` representation.

Syntax: `long * avPValuesINT(` *apl_value* `)`
`AplVal` *apl_value;*

apl_value	Specifies an APL value; the representation type must be `INT`.

Effect: This returns a pointer to the first element of *apl_value*.

avPValuesFLOAT

Purpose: Use `avPValuesFLOAT` to obtain a pointer to the elements of an APL value with `FLOAT` representation.

Syntax: `double * avPValuesFLOAT(` *apl_value* `)`
`AplVal` *apl_value;*

apl_value	Specifies an APL value; the representation type must be `FLOAT`.

Effect: This returns a pointer to the first element of *apl_value*.

avPValuesCHAR

Purpose: Use `avPValuesCHAR` to obtain a pointer to the elements of an APL value with `CHAR` representation.

Syntax: `unsigned char * avPValuesCHAR(` *apl_value* `)`
`AplVal` *apl_value;*

apl_value	Specifies an APL value; the representation type must be `CHAR`.

Effect: This returns a pointer to the first element of *apl_value*.

avPValuesBOOL

Purpose: Use `avPValuesBOOL` to obtain a pointer to the elements of an APL value with `BOOL` representation.

Syntax: `unsigned char * avPValuesBOOL(` *apl_value* `)`
`AplVal` *apl_value;*

apl_value	Specifies an APL value; the representation type must be `BOOL`.

Effect: This returns a pointer to the byte containing the first element of *apl_value*. The elements of a Boolean array, in ravel order, are packed eight to the byte. The elements are arranged within each byte in order of decreasing bit significance. This arrangement does not depend on the byte order of the host machine.

avPValuesPTR

Purpose: Use `avPValuesPTR` to obtain a pointer to the elements of an APL value with `PTR` representation.

Syntax: `AplVal * avPValuesPTR(` *apl_value* `)`
`AplVal` *apl_value;*

apl_value	Specifies an APL value; the representation type of must be `PTR`.

Effect: The result of this function is a pointer to an array of `AplVal`'s.

Managing Storage for APL Values

In most applications, the rank, shape, and representation type of APL values are unknown until execution time. APL interpreters use elaborate schemes for dynamic storage allocation, compaction, and recovery. This section describes machinery that is easy to use and simple to explain. The system allocates the storage for an APL value when it creates the value. A value's reference count can be increased or decreased as pointers to it are created or destroyed. Freeing a value decreases its reference count; if the count reaches 0, the value's storage is released. It is a serious (and difficult to find) error to refer to a value in any way after it has been freed for the last time.

These routines obtain and release storage for APL values via calls to the `AplAlloc` and `AplFree` routines. This storage is temporary in that APL automatically frees it upon return from the external routine. If insufficient memory is available to satisfy an allocation request, these routines do not terminate or signal any kind of error to APL. This is the responsibility of the program requesting the memory allocation. All routines that allocate memory return a null pointer if they are unable to fulfill the requested operation.

Argument values passed to the external routines as *X0* datatype are unusual in that external routines must never free them. APL owns their memory. For protected-mode .REX routines, this memory is actually inside the APL workspace. The system can crash if this memory is freed.

`avAllocate`

Purpose:	Use `avAllocate` to allocate storage and construct a new value.

Syntax:

```
AplVal avAllocate( descriptor, rank, num_elements )
int descriptor;
int rank;
long num_elements;
```

descriptor	Specifies the representation descriptor of the new APL value. Must be one of `CHAR`, `BOOL`, `INT`, `FLOAT`, `PTR`, or `HET`.
rank	Specifies the rank of the new APL value. Must be non-negative and less than 128.
num_elements	Specifies the total number of elements in the new APL value.

Effect: The result is a newly allocated and initialized APL value. The value's rank, descriptor, and number of elements are set. The shape vector is not initialized The caller must ensure that the shape vector of the value is set consistently with the number of elements. New APL values have a reference count of 1. Empty `PTR` values are supplied with a reference to an empty `INT` vector as a prototype. Data elements are not initialized.

`avFree`

Purpose: Use `avFree` to conditionally release the storage used by an APL value created by `avAllocate`.

Syntax:

```
void avFree( x )
AplVal x;
```

x	Specifies the APL value to be destroyed.

Effect: The APL value's reference count is decremented. If the reference count reaches 0, the storage occupied by the value is made available for reuse. If the reference count is not reduced to 0, the value is left intact. The caller must ensure that there is no attempt to use an APL value after its reference count reaches 0.

Nested Arrays

The elements of APL values with `PTR` representation are themselves APL values. Since the opaque C type `AplVal` defined in `aplvals.h` is actually a pointer type, the data part of the `PTR` APL value is an array of pointers to other `AplVal`'s that represent the elements.

If several elements of a `PTR` value are identical, their pointers may point to the same APL value. For example, consider the right argument transmitted to an external routine by the expression:

```
'REX module.Foo(⊂X0)' ⎕NA 'Foo'
Foo (⊂'xy'),4 ρ ⊂'abc'
```

The APL value has five elements and `PTR` representation type. The first element points to a two-element `CHAR` type APL value. The last four elements each point to the same three-element `CHAR` type APL value. Arrays with deeper levels of nesting may have a more complicated structure. The existence of a pointer to an APL value should always be reflected in the value's reference count.

Representation-sharing among identical elements does not happen automatically. The interpreter introduces such representations only when it recognizes the opportunity.

Design your external routines to handle shared elements correctly. Do not assume that representation-sharing will not occur, because future optimizations of the interpreter may take advantage of additional opportunities to create shared elements.

It is generally not worthwhile to introduce representation-sharing unless your external routine returns large nested arrays with many duplicate subroutines.

Remember: Increment a value's reference count when you create a new pointer to it. Decrement a value's reference count when you remove a pointer.

A nested array and its subarrays form a tree of `AplVal` objects. When you transfer a nested array to an external routine, the

APL interpreter packs all of the AplVal objects, along with a record of the pointers among them, into a special representation of the entire tree, where all subarrays occupy contiguous storage. In the APL workspace, the subarrays can reside at any location in the workspace and are seldom contiguous.

The external routine reconstitutes this representation by splitting it into the individual AplVal objects and re-establishing the pointers. When an external routine returns a nested array to APL, it must pack the array and its descendants before transmitting the array to the interpreter. These procedures, which also occur for heterogeneous arrays, are very similar to the procedures that the interpreter uses to store nested arrays in component files.

The avUnpack and avPack routines perform these steps automatically for nested and heterogeneous values. You do not need to add any special code to your external routine to pack or unpack nested values.

When an external routine is called that has an *X*0 datatype as its argument, APL★PLUS II/386 passes a 32-bit pointer (near or far, as described earlier in this section) corresponding to the *X*0 type argument. Your function should declare this argument as type AplVal. The data are passed in "packed external" format; you must unpack it using the avUnpack routine before you can access the value with any other routines.

When an external routine has an *X*0 datatype explicit result, you must return a 32-bit pointer to a "packed external" format value. You can create such a value by calling the avPack routine.

avPack

Purpose: Use `avPack` to prepare an explicit result for return to APL as an *X*0 datatype.

Syntax:

```
AplVal avPack(x)
AplVal x;
```

x Specifies the APL value to be returned.

Effect: The APL value is packed into the format expected by APL★PLUS II/386. This is generally used in the "return" statement as follows:

```
        return acPack( z );
```

This value only formats the data for return to APL. You do not have to retain it.

avUnpack

Purpose: Use `acUnpack` to unpack one APL *X*0 argument value before processing it with other routines. This function can only be called once per argument, because it converts the argument from packed to unpacked format. Calling it twice for the same argument causes unpredictable results.

Syntax:

```
void avUnpack( x )
AplVal x;
```

x Specifies the APL *X*0 argument to be unpacked.

Effect: The APL *X*0 value is converted from packed to unpacked format. After calling this routine, the same pointer that used to refer to the packed format data now points to the unpacked format data. The data do not move in memory; they occupy the same space as the original argument, but the internal pointers are now in unpacked format.

Never attempt to use avFree to free this value, because APL★PLUS II/386 owns the value and releases its memory when you return from the external call.

To avoid problems resulting from accidental freeing of these values, they have very large fictitious reference counts. If the value is nested; that is, if its descriptor is PTR or HET and some item is PTR, then its nested array items are also given large reference counts so that avFree does not deallocate them individually. These inner items are actually allocated in the same storage block as the top-level value and are deallocated when APL★PLUS II/386 frees the top-level value.

Empty Nested Arrays and Prototypes

The definition of the APL★PLUS dialect of APL requires that all APL values have a prototype. For simple values; that is, those represented by CHAR, BOOL, INT, or FLOAT, the prototype is determined by the representation type. Nested arrays are more complicated. For nonempty nested arrays, there is always a first element; that element provides the prototype. For empty nested arrays, it is necessary to maintain a special prototype element. Consequently, APL values with PTR representation type and zero elements always have one element. If you call avAllocate with a descriptor argument of PTR and a number of elements of zero, it is initialized with the default prototype element (an empty integer vector). You can replace this with a different prototype if necessary. A valid prototype element can be an AplVal of any rank, shape, or structure, but all simple items at every level must be zero or space.

Heterogeneous Arrays

For APL values with a representation type of HET, each array element is represented independently and must have its own type descriptor. Each element of a HET value contains sufficient space to represent individual values of any other representation type.

Handling HET arrays is more complicated than handling non-HET arrays. You must test the representation type of each element individually. You cannot move the test outside a per-element loop.

The interpreter imposes an important restriction on HET values: they must need to be heterogeneous. It is an error to create a HET value with elements that are entirely character, numeric, or pointer. You should represent such values by the appropriate type. It is also an error to create an empty or one-element HET value. It is generally inefficient to store or process large HET values unnecessarily.

Accessing the Items of Heterogenous APL Values

The routines described in this section allow you access the items in heterogeneous APL values.

avFetchHetDesc

Purpose: Use avFetchHetDesc to access the descriptor of an item in a heterogenous APL value.

Syntax: int avFetchHetDesc(*apl_value*, *i*)
AplVal *apl_value*;
long *i*;

apl_value	Specifies a heterogenous APL value.
i	Specifies a valid zero origin index into the ravel list of *apl_value*.

Effect: The representation type of the item indexed by *i* is returned. This is a small integer equal to one of CHAR, BOOL, INT, FLOAT, or PTR.

`avFetchHetCHAR`

Purpose:	Use `avFetchHetCHAR` to access the value of a `CHAR` item in a heterogenous APL value.
Syntax:	`unsigned char avFetchHetCHAR(` *apl_value, i* `)` `AplVal` *apl_value*; `long` *i*;
	apl_value Specifies a heterogenous APL value.
	i Specifies a valid zero-origin index into the ravel list of *apl_value*.
Effect:	This function returns the value of the item indexed by *i*. The item must be of type `CHAR`.

`avFetchHetBOOL`

Purpose:	Use `avFetchHetBOOL` to access the value of a `BOOL` item in a heterogenous APL value.
Syntax:	`int avFetchHetBOOL(` *apl_value, i* `)` `AplVal` *apl_value*; `long` *i*;
	apl_value Specifies a heterogenous APL value.
	i Specifies a valid zero origin index into the ravel list of *apl_value*.
Effect:	This function returns the value of the item indexed by *i*. The item must be of type `BOOL`. The result is zero or one.

avFetchHetINT

Purpose: Use `avFetchHetINT` to access the value of an `INT` item in a heterogenous APL value.

Syntax:
`long avFetchHetINT(` ***apl_value, i*** `)`
`AplVal` ***apl_value***;
`long` ***i***;

apl_value	Specifies a heterogenous APL value.
i	Specifies a valid zero origin index into the ravel list of *apl_value*.

Effect: This function returns the value of the item indexed by *i*. The item must be of type `INT`.

avFetchHetFLOAT

Purpose: Use `avFetchHetFLOAT` to access the value of a `FLOAT` item in a heterogeneous APL value.

Syntax:
`double avFetchHetFLOAT(` ***apl_value, i*** `)`
`AplVal` ***apl_value***;
`long` ***i***;

apl_value	Specifies a heterogenous APL value.
i	Specifies a valid zero origin index into the ravel list of *apl_value*.

Effect: This function returns the value of the item indexed by *i*. The item must be of type `FLOAT`.

`avFetchHetPTR`

Purpose: Use `avFetchHetPTR` to access the value of a `PTR` item in a heterogenous APL value.

Syntax:
`AplVal avFetchHetPTR(` *apl_value, i* `)`
`AplVal` *apl_value*;
`long` *i*;

apl_value	Specifies a heterogenous APL value.
i	Specifies a valid zero origin index into the ravel list of *apl_value*.

Effect: This function returns the value of the item indexed by *i*. The item must be of type `PTR`.

`avSetHetCHAR`

Purpose: Use `avSetHetCHAR` to assign a `CHAR` value to an item in a heterogenous APL value.

Syntax: `void avSetHetCHAR(` *apl_value, i, char_value* `)`

`AplVal` *apl_value*;
`long` *i*;
`unsigned char` *char_value*

apl_value	Specifies a heterogenous APL value.
i	Specifies a valid zero origin index into the ravel list of *apl_value*.
char_value	Specifies the character value to be assigned.

Effect: Item *i* of *apl_value* is changed to type `CHAR` and given the value *char_value*.

avSethetBOOL

Purpose: Use `avSetHetBOOL` to assign a `BOOL` value to an item in a heterogenous APL value.

Syntax: `void avSetHetBOOL(` *apl_value*, *i*, *bool_value* `)`

`AplVal` *apl_value*;
`long` *i*;
`int` *bool_value*

apl_value	Specifies a heterogenous APL value.
i	Specifies a valid zero origin index into the ravel list of *apl_value*.
bool_value	Specifies the boolean value to be assigned; should be zero or one.

Effect: Item *i* of *apl_value* is changed to type `BOOL` and given the value *bool_value*.

avSetHetINT

Purpose: Use `avSetHetINT` to assign an `INT` value to an item in a heterogenous APL value.

Syntax: `void avSetHetINT(` *apl_value*, *i*, *long_value* `)`

`AplVal` *apl_value*;
`long` *i*;
`long` *long_value*

apl_value	Specifies a heterogenous APL value.
i	Specifies a valid zero origin index into the ravel list of *apl_value*.
long_value	Specifies the long value to be assigned.

Effect: Item *i* of *apl_value* is changed to type `INT` and given the value *long_value*.

avSetHetFLOAT

Purpose: Use `avSetHetFLOAT` to assign a `FLOAT` value to an item in a heterogenous APL value.

Syntax: `void avSetHetFLOAT(` *apl_value*, *i*, *double_value* `)`

`AplVal` *apl_value*;
`long` *i*;
`double` *double_value*

apl_value	Specifies a heterogenous APL value.
i	Specifies a valid zero origin index into the ravel list of *apl_value*.
double_value	Specifies the double value to be assigned.

Effect: Item *i* of *apl_value* is changed to type `FLOAT` and given the value *double_value*.

avSetHetPTR

Purpose: Use `avSetHetPTR` to assign a `PTR` value to an item in a heterogenous APL value.

Syntax: `void avSetHetPTR(` *apl_value*, *i*, *ptr_value* `)`

`AplVal` *apl_value*;
`long` *i*;
`AplVal` *ptr_value*

apl_value	Specifies a heterogenous APL value.
i	Specifies a valid zero origin index into the ravel list of *apl_value*.
ptr_value	Specifies the APL value to be assigned.

Effect: Item *i* of *apl_value* is changed to type `PTR` and given the value *ptr_value*.

Convenience Routines

The routines in this section extend the basic value manipulation capabilities described earlier. They make it more convenient to manipulate APL values from your external routine.

`avHetBoolToInt`

Purpose: The function `avHetBoolToInt` coerces all top-level Boolean values in a heterogenous object to integer representation.

Syntax:

```
void avHetBoolToInt( apl_value )
AplVal apl_value;
```

apl_value	Specifies an APL value.

Effect: If *apl_value* has `HET` representation, any top-level Boolean elements are changed to the corresponding `INT` representation. Non-`HET` arguments are not changed. This facilitates the handling of elements that are always treated as integers, but may occasionally have `BOOL` representation.

`avMakeCharVector`

Purpose: Use `avMakeCharVector` to construct an APL character vector value from a null-terminated string.

Syntax:

```
AplVal avMakeCharVector( string, nelm )
char FAR * string;
long nelm;
```

string	Specifies elements of result.
nelm	Specifies the number of elements.

Effect: The result of this function is a newly constructed APL value. It has `CHAR` representation, rank one, and as many elements as null-terminated character string (if *nelm*=-1) or as many as specified by *nelm* (if *nelm* ≥0). If *nelm* ≥0, then *string* can be NULL.

`avMakeIntVector`

Purpose: Use `avMakeIntVector` to construct an APL integer vector.

Syntax:

```
AplVal avMakeIntVector( values, nelm )
long FAR * values;
long nelm;
```

values	Specifies elements of result.
nelm	Specifies the number of elements.

Effect: The result of this function is a newly constructed APL value. It has `INT` representation, rank one, and the number of elements specified by *nelm*. If *values* is not NULL, the array is initialized with *nelm* values starting at the address specified by *values*.

`avMakeFloatVector`

Purpose: Use `avMakeFloatVector` to construct an APL floating-point vector.

Syntax:

```
AplVal avMakeFloatVector( values, nelm )
long FAR * values;
long nelm;
```

values	Specifies elements of result.
nelm	Specifies the number of elements.

Effect: The result of this function is a newly constructed APL value. It has `FLOAT` representation, rank one, and the number of elements specified by *nelm*. If *values* is not NULL, the array is initialized with *nelm* values starting at the address specified by *values*.

avMakeIntScalar

Purpose: Use auMakeIntScalar to construct an APL integer scalar value from a C long.

Syntax: AplVal auMakeIntScalar(*long_value*)
long ***long_value***;

long_value Specifies the single element of the result.

Effect: The result of this function is a newly constructed APL value. It has INT representation, rank zero, and one element.

avMakeFloatScalar

Purpose: Use avMakeFloatScalar to construct an APL floating-point scalar value from a C long.

Syntax: AplVal avMakeFloatScalar(*long_value*)
long ***long_value***;

long_value Specifies the single element of the result.

Effect: The result of this function is a newly constructed APL value. It has FLOAT representation, rank zero, and one element.

avMakeCharScalar

Purpose: Use avMakeCharScalar to construct an APL character scalar value from a C long.

Syntax: AplVal avMakeCharScalar(*long_value*)
long ***long_value***;

long_value Specifies the single element of the result.

Effect: The result of this function is a newly constructed APL value. It has CHAR representation, rank zero, and one element.

`avFetchLong`

Purpose: You can use `avFetchLong` to fetch and coerce an integer value from an APL value.

Syntax: `long avFetchLong(` *apl_value*, *i* `)`
`AplVal` *apl_value*;
`long` *i*;

apl_value Specifies an APL value.

i Specifies a zero origin index into the ravel list of *apl_value*.

Effect: This function attempts to fetch the element of *apl_value* indexed by *i*. The result is always of type `long`. `BOOL` elements are promoted. `FLOAT` elements are rounded but not checked against the range of a `long`. The integer specified by the *aplerror* external variable is set to 1 if an error occurs. If an error occurs, you must clear this setting back to 0 before calling this routine. This routine sets *aplerror* by calling `AplError` to signal an error to APL. You can cancel the error signal to APL by calling `AplError` with a NULL argument.

`avFetchBool`

Purpose: You can use `avFetchBool` to fetch a Boolean value from an APL value.

Syntax: `long avFetchBool(` *apl_value*, *i* `)`
`AplVal` *apl_value*;
`long` *i*;

apl_value Specifies an APL value.

i Specifies a zero origin index into the ravel list of *apl_value*.

Effect: This function attempts to fetch the element of *apl_value* indexed by *i*. The index is checked, `FLOAT` elements are rounded, and the status is returned as for `avFetchLong`. The integer specified by the *aplerror* external variable is set to zero if the function is successful. It is set to 1 if an error occurs. If an error occurs, you must clear this setting back to 0 before calling this routine. This routine sets *aplerror* by calling `AplError` to signal an error to APL. You can cancel the error signal to APL by calling `AplError` with a NULL argument.

avSetBool

Purpose: You can use `avSetBool` to assign to an element in an APL value with `BOOL` representation type.

Syntax:

```
long avSetBool( apl_value, i, b )
AplVal apl value;
long i;
int b;
```

apl_value	Specifies an APL value of `BOOL` representation type.
i	Specifies a zero origin index into the ravel list of *apl_value*.
b	Specifies the value to be assigned. Should be zero or one.

Effect: This function attempts to assign to the element of *apl_value* indexed by *i*. The integer specified by the *aplerror* external variable is set to zero if the function is successful. It is set to 1 if an error occurs. If an error occurs, you must clear this setting back to 0 before calling this routine. This routine sets *aplerror* by calling `AplError` to signal an error to APL. You can cancel the error signal to APL by calling `AplError` with a NULL argument. It is an error to call this function if *apl_value* does not have a `BOOL` representation type.

avCoerce

Purpose:	Use this routine to coerce an APL value from one type to another.

Syntax:	`AplVal avCoerce( x, descriptor )`
	`AplVal x;`
	`int descriptor;`

Effect:	Converts an APL value from one type to another with same rank and shape as the original array. This function is similar to the effect of `avAllocate`, except that the array is filled in with converted values from the argument array.
Note:	This routine does not report conversion errors. You can truncate values or convert them from `CHAR` to non-`CHAR`. You cannot convert nested arrays into simple arrays. You cannot use `PTR` as a target array type. You must use `HET` instead, and let APL demote the `HET` to `PTR`.

avAllocEmptyMat

Purpose:	Use this routine to create an empty matrix.
Syntax:	`AplVal AvAllocEmptyMat( void );`
Effect:	Creates an empty integer or character matrix.

avAllocEmptyVec

Purpose:	Use this routine to create an empty vector.
Syntax:	`AplVal AvAllocEmptyVec( void );`
Effect:	Creates an empty integer vector.

avAllocEmptyCharVec

Purpose: Use this routine to create an empty vector.

Syntax: `AplVal AvAllocEmptyCharVec( void );`

Effect: Creates an empty character vector.

Index

READER COMMENT CARD

We welcome your evaluation of this manual. Your comments and suggestions help us improve our publications.

Product name: **APL★PLUS II/386** Software version number ____________________

Title of the manual you are evaluating: *User Manual*

Please circle one number for each.	**Strongly Agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
• The manual is well organized.	1	2	3	4	5
• The manual is easy to understand.	1	2	3	4	5
• The manual is complete.	1	2	3	4	5
• The manual is clearly written.	1	2	3	4	5
• I can find the information I want.	1	2	3	4	5
• Concepts and vocabulary are easy to understand.	1	2	3	4	5
• Examples are clear and useful.	1	2	3	4	5
• The manual contains enough examples.	1	2	3	4	5
• Illustrations are clear and helpful.	1	2	3	4	5
• The manual contains enough illustrations.	1	2	3	4	5
• The index is thorough.	1	2	3	4	5
• Layout and format enhance the manual's usefulness.	1	2	3	4	5
• This manual meets my overall expectations.	1	2	3	4	5

(over, please)

Please write additional comments, particularly if you disagree with a statement above. Use additional pages if you wish. The more specific your comments, the more useful they are to us.

Comments: ______________________________

____

Although optional, we would appreciate the following information.

Name______________________________

Title______________________________

Company______________________________

Address______________________________

City/State/Zip______________________________

Country______________________________

Phone ______________________________

STSC

STSC, Inc.
2115 East Jefferson Street
Rockville, MD 20852
U.S.A.

ISBN 0-926683-36-5